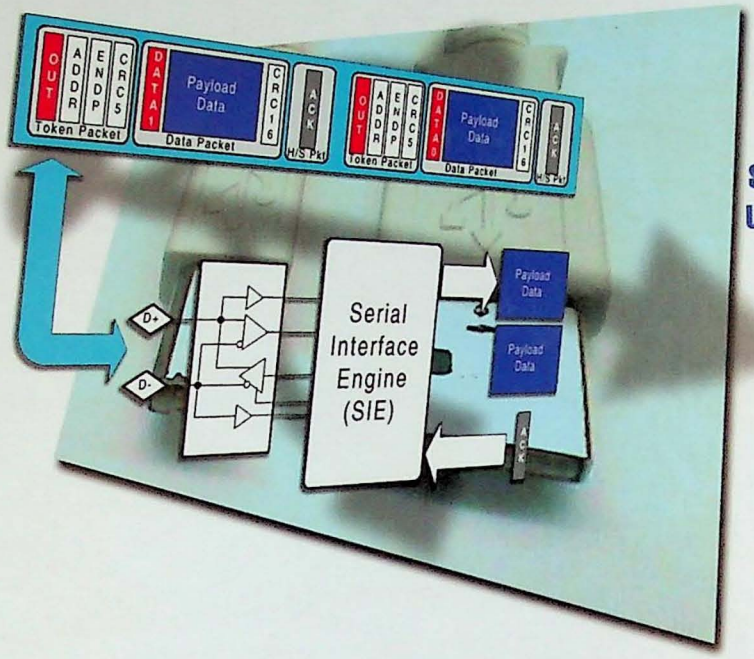
PC & Elektronik

Burkhard Kainka

USB-Interfaces mit analogen und digitalen Ein- und Ausgängen

Bauteile, Schaltungstechnik, Messtechnik

Software für PC und USB-Mikrocontroller



Messen, Steuern und Regeln mit USB

Hard- und Software-Entwicklung für die Praxis

Auf CD-ROM

- MSR-Anwenderprogramme
- Infos zu USB-Komponenten
- Quelltexte

Franzis'

Kainka

Messen, Steuern und Regeln mit USB

PC & Elektronik

Burkhard Kainka

Messen, Steuern und Regeln mit USB

Hard- und Software Entwicklung für die Praxis

Mit 111 Abbildungen

Franzis'

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist bei
Der Deutschen Bibliothek erhältlich

© 2000 Franzis Verlag GmbH, 85586 Poing

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.

Die meisten Produktbezeichnungen von Hard- und Software sowie Firmennamen und Firmenlogos, die in diesem Werk genannt werden, sind in der Regel gleichzeitig auch eingetragene Warenzeichen und sollten als solche betrachtet werden. Der Verlag folgt bei den Produktbezeichnungen im wesentlichen den Schreibweisen der Hersteller.

Satz: Conrad Neumann, 81827 München
Druck und Bindung: Kösel, Kempten (www.koeselbuch.de)
Printed in Germany - Imprimé en Allemagne.

ISBN 3-7723-5874-8

Vorwort

Seit der Einführung von Windows 98 ist der Universelle Serielle Bus (USB) eine wichtige Schnittstelle jedes modernen PCs. Viele Nachteile der bisherigen Schnittstellen-Vielfalt werden damit aufgehoben. Langfristig hat der USB die Chance, die meisten der bisherigen Schnittstellen abzulösen.

Was für den reinen PC-Anwender eine Erleichterung ist, kann für den engagierten Hobbyanwender und für manchen Entwickler zu einem Problem werden. Auf den ersten Blick scheint es nämlich, als wäre der USB nicht mehr für eigene Hardware-Entwicklungen geeignet. Sehr einfache Interfaces, wie sie für die serielle Schnittstelle leicht aufgebaut werden konnten, gibt es beim USB nicht. Der neue Bus ist komplexer und aufwendiger.

Mit diesem Buch wird der Versuch unternommen, den USB auch für Hobbyanwendungen nutzbar zu machen. Es werden einfache Zugänge gesucht und Hilfestellungen gegeben. Das Buch stellt beispielhaft Entwicklungen einfacher Interfaces für den USB vor. Dabei werden alle Aspekte von der Controller-Programmierung über die Windows-Software bis zu Treibern beleuchtet. Auf der CD findet man die wichtigsten Werkzeuge und alle vorgestellten Beispielprogramme des Buchs.

Ich hoffe, mit diesem Buch vielen Lesern den Einstieg in die Arbeit mit dem USB zu erleichtern.

Burkhard Kainka, Essen
http://home.t-online.de/home/B.Kainka

Inhalt

1 Einleitung ... 11

2 USB-Grundlagen ... 13
2.1 Anschlüsse und Kabel ... 13
2.2 Serielle Übertragung ... 15
2.3 Hubs ... 18
2.4 USB-Transfertypen ... 20
2.5 Enumeration ... 21
2.6 Treiber-Aufrufe ... 27
2.7 CreateFile und CloseFile ... 29
2.8 WriteFile und ReadFile ... 30
2.9 DeviceIoControl ... 31

3 USB-Standardgeräte ... 36
3.1 Installation einer USB-Maus ... 36
3.2 Eine USB-Soundkarte ... 38
3.3 Der Signalgenerator AUDIO-Wave ... 42
3.4 Ein Gehörtest ... 44
3.5 Ein USB-Joystick-Port ... 47

4 Stromversorgung aus dem USB-Kabel ... 54
4.1 Belastung und Innenwiderstand ... 54
4.2 Absicherung ... 56
4.3 Relaisanschluss ... 56
4.4 Spannungsstabilisierung ... 57
4.5 Messungen an den Signalleitungen ... 58

5 Der USB-Controller CY7C63000 ... 60
5.1 Das USB-Thermometer von Cypress ... 62
5.2 Programmierung in Visual Basic ... 64
5.3 Einstellung des LED-Stroms ... 66
5.4 Weitere Treiberfunktionen ... 69
5.5 Analyse der USB-Datenübertragung ... 70
5.6 Portausgaben ... 75

6 Ein universelles USB-Interface ... 79
6.1 Der AD-Wandler TLC1543 ... 79
6.2 Zusammenfassung von AD-Kanälen ... 86
6.3 AD-Eingänge als Digitaleingänge ... 91
6.4 Gemeinsame Übertragung der Nutzdaten ... 98
6.5 Programmierung mit Delphi ... 101
6.6 Elektrische Eigenschaften der Interface-Ports ... 104
6.7 Stromversorgung ... 107
6.8 Das CompuLAB-USB ... 109

7 Treiber ... 114
7.1 Umbau eines Beispiels-Treibers ... 114
7.2 Anpassung der Firmware ... 122
7.3 Die INF-Datei ... 123

8 Messungen und Experimente ... 126
8.1 Das Anwenderprogramm Compact 2000 ... 126
8.2 Die Programmierumgebung in Compact 2000 ... 130

9 Der USB-Controller AN2131 ... 138
9.1 Technische Daten ... 138
9.2 Das EZ-USB-Starterkit ... 141
9.3 Das Default Anchor Device ... 143
9.4 Treiberaufrufe in Delphi ... 145

10 Ein Fullspeed-USB-Interface ... 148
10.1 Schaltung und Aufbau ... 148
10.2 Lesen von Portzuständen ... 151
10.3 Portausgaben ... 154
10.4 Basisfunktionen für USB-Zugriffe ... 158
10.5 Ein Logik-Analysator ... 163

11 Der AD-Wandler MAX186 ... 168
11.1 Anschlüsse und Betriebsarten ... 169
11.2 Das Anwenderprogramm Serai8/12 USB ... 171
11.3 Assembler-Routine zum AD-Wandler ... 174
11.4 Ein Speicher-Oszilloskop ... 181
11.5 Triggerung ... 186

12 Der I²C-Bus ... 192
12.1 Das Busprotokoll ... 192
12.2 Steuerregister ... 194

12.3 Portexpander PCF8574 196
12.4 I²C-EEPROMs 204
12.5 Der EEPROM-Bootloader 216

13 Bulk-Transfer 220
13.1 Pipes und Endpoints 220
13.2 Ein Assembler-Programm zum Bulk-Transfer 222
13.3 Host-Software zum Bulk-Transfer 224
13.4 Portzugriffe über Bulk-Transfers 230

14 Anhang 235
14.1 Die Delphi-Unit EZUSB2.PAS 235
14.2 Register des AN2131 245
14.3 Literatur 249
14.4 Bezugsadressen 249

Sachverzeichnis 250

1 Einleitung

Für Messgeräte und Interfaces in Labor-Anwendungen wurde bisher vielfach die RS232-Schnittstelle verwendet. Ein Umstieg auf den USB bringt Vorteile aber auch Schwierigkeiten mit sich.

Der Universal Serial Bus erleichtert die Arbeit für den Anwender im Allgemeinen und den Einsatz im Labor im Speziellen. Zunächst besitzt der USB eine größere Bandbreite als die serielle Schnittstelle. In der USB-Version 1.1 gibt es Lowspeed-Geräte mit 1,5 Mb/s und Fullspeed-Geräte mit 12 Mb/s. Bereits die kleinere Übertragungsgeschwindigkeit überragt die möglichen Baudraten an der seriellen Schnittstelle um ein Vielfaches. Bei einer Gegenüberstellung beider Schnittstellen muss aber auch berücksichtigt werden, dass ein einzelnes Gerät niemals die volle Bandbreite beanspruchen kann.

Bisher passierte es oft, dass alle Schnittstellen des PC bereits belegt waren. Der USB bringt hier den Vorteil, dass mit dem Anschluss eines zusätzlichen Busverteilers (Hub) wieder drei neue Ports freistehen. Insgesamt können bis zu 127 Geräte an den USB angeschlossen werden. Wenn also ein bestehendes Messsystem um einige Kanäle erweitert werden soll, ist das am USB prinzipiell möglich.

Der USB liefert gleich auch die Betriebsspannung für kleinere Geräte mit. Bis zu 100 mA können ohne weitere Probleme entnommen werden. Typische Laboranwendungen kommen oft mit wesentlich weniger aus. Der Wegfall eines zusätzlichen Versorgungskabels pro Gerät hilft im Kampf gegen den allgemeinen Kabelsalat.

USB-Geräte können grundsätzlich im laufenden Betrieb angeschlossen und getrennt werden. Das Betriebssystem lädt automatisch den erforderlichen Treiber. Diese erweiterte Plug-And-Play-Fähigkeit erleichtert den Umgang mit mehreren Geräten erheblich.

Die Anwendung des USB ist hier auf PCs mit dem Betriebssystem Windows 98 eingeschränkt. Unter Windows 95 ist USB zwar ab der B-Version prinzipiell möglich. Jedoch wird hier noch nicht die volle Zuverlässigkeit erreicht. Für die Arbeit mit diesem Buch wird daher Windows 98 empfohlen. Windows NT 4 unterstützt den USB nicht. Erst NT 5 bzw. Windows 2000 verwendet den USB. Allerdings fehlen teilweise noch die erforderlichen Treiber. Speziell die Anwendungen in diesem Buch wurden nur mit Windows98 getestet.

Ein Nachteil für den Hobbyanwender ist die große Komplexität des USB. Einfache Geräte wie an der RS232 sind hier nicht möglich. Ein USB-System verfügt mindestens über einen USB-fähigen Mikrocontroller mit einem umfangreichen Programm. Auf der PC-Seite ist ein Treiber erforderlich. Auch das Schreiben eines Treibers ist nicht einfach.

Wer ernsthafte Entwicklungen betreiben will, der steht noch vor einer weiteren Hürde. Jedes USB-Gerät verfügt über eine interne Hersteller-Nummer (Vendor-ID), die offiziell von der USB-Organisation vergeben wird. Ein Gerät kann also nur mit einer gültigen VID auf den Markt gebracht werden.

Dieses Buch versucht das Thema USB auch für den engagierten Hobbyanwender zugänglich zu machen. Es wird daher zunächst von im Handel erhältlichen USB-Produkten ausgegangen. Vielfach sind auch hier schon einfache Versuche möglich.

Einige Hersteller integrierter Bausteine für den USB unterstützen die Entwicklung durch Starter-Kits und im Internet veröffentlichte Beispielanwendungen. Sie ermöglichen den ersten Kontakt mit der Materie und unterstützen eigene Entwicklungen. Dieses Buch stützt sich auf Vorlagen der Halbleiterhersteller und zeigt, wie sich in kleinen Schritten eigene USB-Geräte entwickeln lassen. Beispielhaft wird in die Arbeit mit den entsprechenden Mikrocontrollern eingeführt. Zugleich werden die erforderlichen Grundlagen der Programmierung auf der PC-Seite gelegt. Als Programmiersprachen kommen Visual Basic und Delphi zum Einsatz.

2 USB-Grundlagen

Der USB ist ein serieller Bus. Daten werden also ähnlich wie bei der RS232 bitweise nacheinander übertragen. Allerdings laufen sie auf denselben Leitungen in beiden Richtungen, während es bei der RS232 getrennte Leitungen für beide Richtungen gibt. Der wesentliche Unterschied liegt aber darin, dass die bisherige serielle Schnittstelle immer nur ein Gerät verwenden konnte, der USB dagegen bis zu 127. Jedes angeschlossene Gerät erhält eine Adresse und kann damit gezielt angesprochen werden.

Der USB ist wesentlich schneller als die RS232. Daten können mit bis zu 12 Megabit pro Sekunde über die Signalleitungen geführt werden. Diese Bandbreite teilt sich allerdings auf alle am Bus angeschlossenen Geräte auf.

Der USB ist eng mit dem Begriff Plug-And-Play verbunden. Geräte können im laufenden Betrieb angeschlossen oder entfernt werden. Das System erkennt ein neues Gerät und lädt automatisch den entsprechenden Treiber.

Ein weiteres wesentliches Merkmal des USB ist, dass auch die Betriebsspannung weitergeleitet wird. Kleinere Geräte kommen also ohne eigenes Netzteil aus.

2.1 Anschlüsse und Kabel

Der USB kennt zwei verschiedene Steckertypen, Typ A und Typ B. Das System ist so konzipiert, dass keine Verwechslung möglich ist. Anders als bei der RS232 gibt es auch keine Unterschiede zwischen gekreuzten und geraden Verbindungen. Kabel sind immer 1:1 verbunden. Die Belegung ist immer gleich:

1 +5V
2 Data -
3 Data +
4 Masse

An der Rückseite eines modernen PCs findet man zwei Buchsen vom Typ A. Hier können direkt zwei Geräte angeschlossen werden. Kleinere Lowspeed-Geräte wie z.B. eine Maus verwenden ein dünnes fest angebrachtes Kabel mit einem Stecker

vom Typ A. In anderen Fällen enthält das Gerät selbst eine USB-Buchse vom Typ B. Die Verbindung erfolgt dann mit einem Kabel vom Typ A-B.

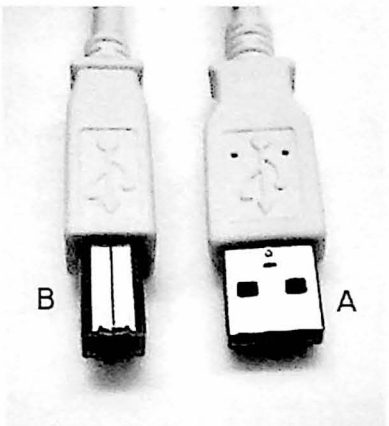


Abb. 2.1: Die verwendeten Stecker

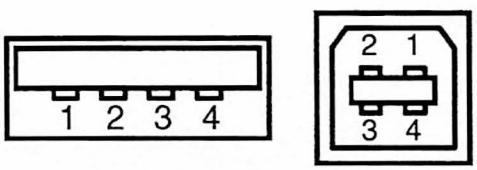


Abb. 2.2: Nummerierung der Anschlüsse an den Buchsen

Solche Kabel gibt es nur fertig konfektioniert und vergossen. Einzelne USB-Stecker sind nicht erhältlich. Länge, Kabelquerschnitt, Abschirmung usw. sind genau vorgeschrieben. Auch der Unterschied zwischen Full-Speed und Low-Speed spielt hier eine Rolle. Das System der vorgeschriebenen Kabel verhindert zuverlässig, dass ein Lowspeed-Kabel für eine Fullspeed-Verbindung eingesetzt wird. Alle Verbindungskabel sind Fullspeed-Kabel, während Lowspeed-Kabel nur fest angebaut vorkommen. Für den Fall einer notwendigen Kabelverbindung gibt es Verlängerungskabel vom Typ A-A.

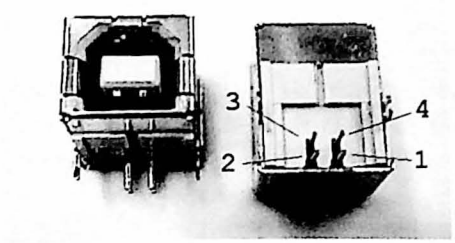


Abb. 2.3: USB-Buchsen vom Typ B

Während USB-Stecker nicht einfach zu bekommen sind, gibt es einzelne USB-Buchsen für die Platinenmontage (vgl. Abb. 2.3). Eigene Experimente sind also durchaus möglich. Der Anschluss stellt eine Betriebsspannung von +5 V zu Verfügung, die ohne weiteres bis 100 mA belastet werden kann. Die beiden Datenleitungen lassen sich nur in Zusammenhang mit speziellen USB-Bausteinen, also z.B. mit speziellen Mikrocontrollern nutzen. Diese können dann direkt aus dem USB-Anschluss versorgt werden. Nach einer besonderen Anmeldung darf ein Gerät bis zu 500 mA entnehmen.

Die Signale auf den beiden Leitungen D+ und D- sind Differenzsignale mit Spannungspegeln von 0V/3,3V. Der angeschlossene Mikrocontroller arbeitet in vielen Fällen mit einer Betriebsspannug von 3.3 V. Die Versorgungsspannung am USB kann bis zu 5,25 V betragen und bei starker Belastung bis auf 4,2 V abfallen. Ein Spannungsregler liefert auch in diesem Fall noch stabile 3,3 V. Das gesamte System ist so ausgelegt, dass auch bei maximaler Belastung eine Versorgungsspannung von 4,2 V nicht unterschritten wird. Niemand soll daher eigene Kabel einsetzen, die einen höheren Widerstand haben könnten. Geräte die mehr als 100 mA benötigen, müssen ihren Bedarf beim System anmelden und werden nur dann zugelassen, wenn noch genügend Reserven vorhanden sind.

Man unterscheidet zwischen Geräten mit eigenem Netzteil (Self-Powered) und solchen, die über den USB versorgt werden (Bus-Powered). In vielen Fällen können beide Betriebsarten gewählt werden. Das Gerät besitzt dann z.B. eine Netzteilbuchse, die wahlweise mit einem externen Netzteil verbunden werden kann. Nach den USB-Spezifikationen ist die Stromaufnahme aus dem Bus automatisch begrenzt. Wird also mehr als der erlaubte Strom entnommen, soll die Versorgung abgeschaltet werden.

2.2 Serielle Übertragung

Der USB ist ein Bus mit nur einem Master, d.h. alle Aktivitäten gehen vom PC aus. Daten werden in kurzen Paketen von 8 Bytes oder bis zu 256 Bytes versandt und empfangen. Der PC kann Daten von einem Gerät anfordern. Umgekehrt kann kein Gerät von sich aus Daten absenden.

Der gesamte Datenverkehr hat einen Rahmen von exakt einer Millisekunde. Innerhalb eines Rahmens können nacheinander Datenpakete für mehrere Geräte verarbeitet werden. Dabei können Lowspeed- und Fullspeed-Pakete zusammen vorkommen. Wenn mehrere Geräte angesprochen werden, sorgt ein Bus-Verteiler (Hub) für die Verteilung. Er verhindert auch, dass Fullspeed-Signale an Lowspeed-Geräte weitergeleitet werden.

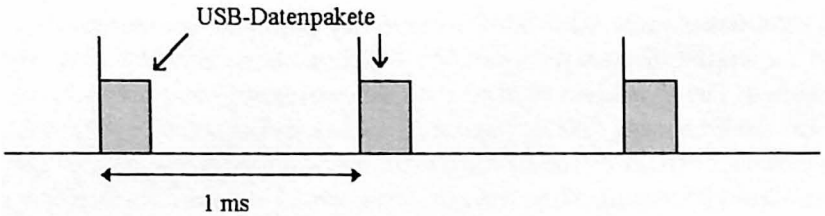


Abb. 2.4: Datenpakete in 1-ms-Frames

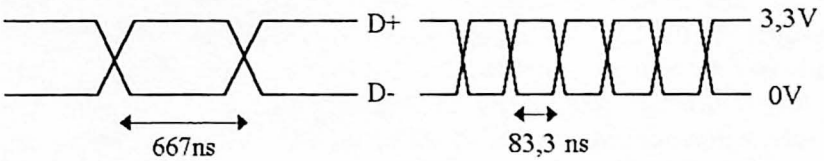


Abb. 2.5: Lowspeed- und Fullspeed-Signale

Lowspeed-Geräte arbeiten mit einer Datenrate von 1,5 Mb/s, d.h. ein Bit ist exakt 666,7 ns lang. Fullspeed-Verbindungen verwenden 12 Mb/s bzw. 83,33 ns. Die Geschwindigkeit wird allein vom Master vorgegeben. Die Slaves müssen sich auf den Datenstrom synchronisieren. Da kein getrenntes Taktsignal übertragen wird, muss der Takt aus dem Datensignal gewonnen werden. Man verwendet dazu das NRZI-Verfahren (Non-Return-to-Zero). Daten-Nullen führen hierbei zu einem Pegelwechsel, Einsen lassen den Pegel unverändert.

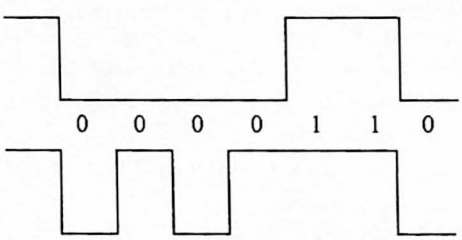


Abb. 2.6: NRZI-Signal

Die Kodierung und Dekodierung der Signale ist allein Sache der Hardware. Der Empfänger muss den Takt zurückgewinnen, die Daten empfangen und dekodieren. Ein besonderes Verfahren sorgt dafür, dass die Synchronisierung nicht verloren geht. Wenn der ursprüngliche Datenstrom sechs folgende Einsen enthält, wird vom Sender automatisch eine Null eingefügt (Bit-Stuffing), um einen Pegelwechsel zu erzwingen. Der Empfänger entfernt diese Null wieder aus dem Datenstrom. Jedes Datenpaket hat zum Zweck der Synchronisation einen besonderen Vorspann, das Sync-Byte (00000001b). Der Empfänger sieht wegen der NRZI-Kodierung und dem

Bit-Stuffing acht wechselnde Bitzustände, auf die er sich synchronisieren kann. Während der folgenden Übertragung muss die Synchronisierung erhalten bleiben. All diese Vorgänge laufen allein in entsprechenden Hardware-Bausteinen ab, die ähnlich wie ein UART für die RS232 die eigentliche Arbeit übernehmen. Man sendet also am einen Ende Daten ab, die am anderen Ende wieder erscheinen. Was dazwischen passiert, muss man zum Glück nicht so genau verstehen.

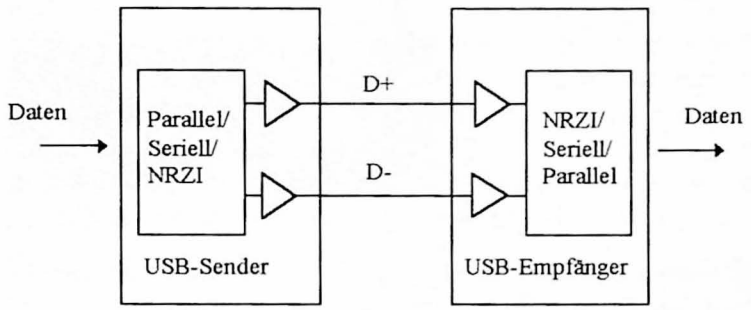


Abb. 2.7: USB-Sender und Empfänger

Empfänger und Sender werden immer zusammen in einem Baustein realisiert. Ein USB-Gerät enthält eine Serial Interface Engine (SIE), die die eigentliche Arbeit übernimmt. Zum Datenaustausch zwischen der SIE und dem Rest des Geräts dienen FIFO-Puffer. FIFOs (First In First Out) sind Speicher, die ähnlich einem Schieberegister Daten sequentiell aufnehmen und abgeben können. Ein angeschlossener Mikrocontroller braucht also nur Daten aus einem FIFO zu lesen und andere Daten in ein FIFO zu schreiben. Alles andere erledigt die SIE. In den meisten Fällen ist die SIE ein Bestandteil des USB-Mikrocontrollers. Es gibt jedoch auch Einzelbausteine wie z.B. den PDIUSBD11 und den PDIUSBD12 von Philips, die sich in ein bestehendes Mikrocontroller-System ähnlich wie ein UART einfügen lassen.

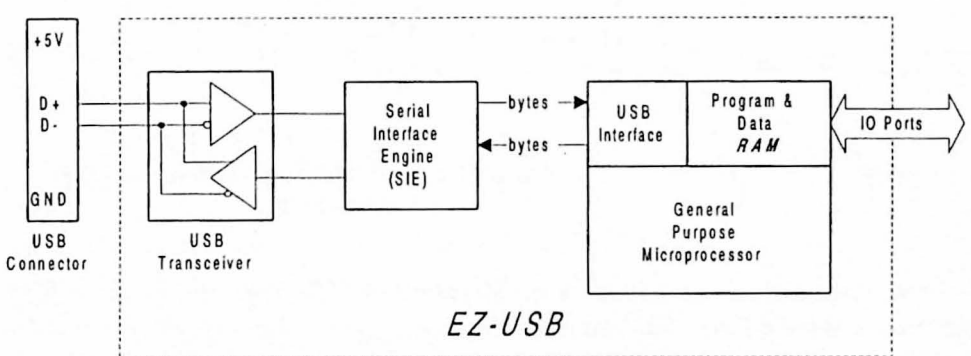


Abb. 2.8: Die SIE im Mikrocontroller AN2131

Ein USB-Gerät hat im allgemeinen mehrere FIFO-Speicher, an die Daten übertragen werden können. Zur Geräteadresse kommt noch eine sog. Endpoint-Adresse, die angibt, wohin die Daten gelangen sollen, oder von wo sie geholt werden sollen. So hat z.B. eine USB-Maus immer einen Endpoint 0 und einen Endpoint 1. Endpoint 0 wird während der Initialisierung verwendet. Die eigentlichen Nutzdaten werden vom Mikrocontroller in gewissen Abständen in den Endpoint-1-FIFO geschrieben und dort vom PC abgeholt.

Die USB-Software bildet sogenannte Pipes zu den einzelnen Endpoints. Eine Pipe ist ein logischer Kanal zu einem Endpoint in einem Gerät. Man kann sich eine Pipe wie einen Datenkanal vorstellen, der durch einen einzelnen Draht gebildet wird. Tatsächlich aber werden die Daten einer Pipe als Datenpakete in einem Millisekunden-Frame übertragen und von der Hardware anhand ihrer Endpoint-Adresse auf reale Speicher verteilt. Ein Gerät kann mehrere Pipes gleichzeitig verwenden, so dass die Datenrate insgesamt ansteigt.

2.3 Hubs

Der USB ist ein sternförmiger Bus mit einem Master. Zum Anschluss mehrerer USB-Peripheriegeräte benötigt man einen Hub. Ein Hub ist ein Bus-Verteiler mit mehreren Ports. Der Name kommt vom englischen Wort „hub" für „Radnabe" und beschreibt bildlich die Verbindung von Speichen eines Rades. Üblich ist ein externer Hub mit einem Upstream-Port und vier Downstream-Ports. Aber der PC selbst enthält schon einen Hub, um zwei USB-Anschlüsse zu realisieren. Dieser sog. Root-Hub befindet sich auf dem Motherboard.

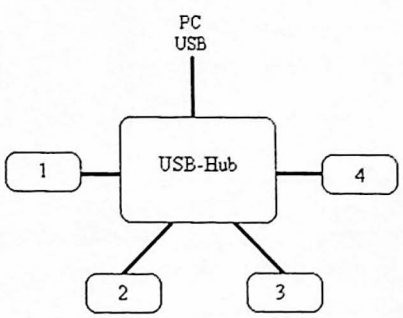


Abb. 2.9: Der sternförmige Aufbau des USB mit einem Hub

Am Downstream-Port eines Hubs kann ein weiterer Hub angeschlossen werden. Jeder Hub und jedes Kabel führt zu einer Verzögerung von Signalen, die einen definierten Maximalwert nicht überschreiten darf. Insgesamt dürfen bis zu sieben Hubs

hintereinander liegen. Daraus ergibt sich die maximale Anzahl von 127 Geräten. Dies ist ein eher theoretischer Wert, da die gesamte Bus-Bandbreite auf alle angeschlossenen Geräte verteilt werden muss.

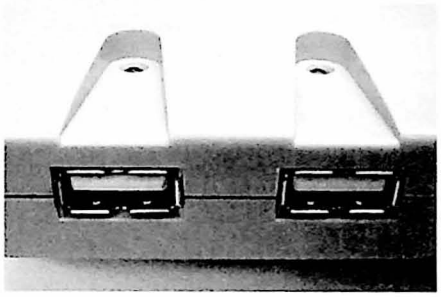


Abb. 2.10: Zwei Downstream-Ports eines Hub

Eine Aufgabe des Hub ist die Erkennung neu angeschlossener Geräte. Er muss erkennen, ob es sich um ein Fullspeed- oder um ein Lowspeed-Gerät handelt. Außerdem kann er in einem angeschlossenen Gerät einen Reset auslösen. Dazu gibt es besondere Bus-Zustände.

Ein nicht verwendeter USB-Anschluss ist nicht aktiv, d.h. der Hub sendet keinen Datenrahmen. Beide Signalleitungen sind low und haben einen Innenwiderstand von 15 kΩ. Jedes USB-Peripheriegerät besitzt einen Widerstand von 1,5 kΩ, der eine der beiden Signalleitungen mit +3,3 V verbindet. Bei einem Fullspeed-Gerät wird D+ hochgezogen (vgl. *Abb. 2.11*), bei einem Lowspeed-Gerät D- (vgl. *Abb. 2.12*). Der Hub erkennt also den Typ des Geräts und kann eine Datenverbindung mit der entsprechenden Übertragungsrate aufbauen.

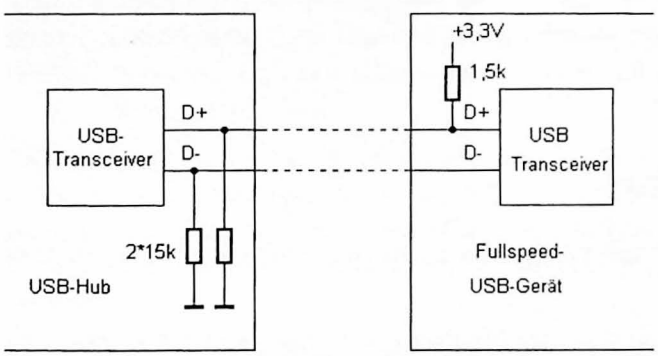


Abb. 2.11: Erkennung eines Fullspeed-Geräts

Als erstes wird ein Bus-Reset ausgeführt, indem beide Datenleitungen für 10 ms aktiv heruntergezogen werden. Die SIE des Peripheriegeräts erkennt diesen Zustand und löst einen Reset des angeschlossenen Mikrocontrollers aus. Dieser beginnt dann mit der Abarbeitung seines Controller-Programms (Firmware) und ist bereit für die Anmeldung (Enumeration) am System.

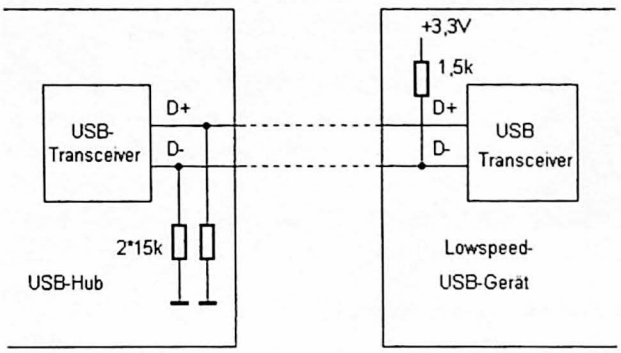


Abb. 2.12: Erkennung eines Lowspeed-Geräts

Der Hub versorgt die Geräte mit Betriebsspannung. Beim Start darf jedes USB-Gerät bis zu 100 mA aufnehmen. Falls ein höherer Bedarf besteht, muss dieser angemeldet werden. Der Hub kann dann mehr Strom erlauben. Entsprechende Überstrombegrenzungen, z.B. durch Polyswitch-Sicherungen, sollen die Stromaufnahme begrenzen. Der Hub soll entsprechende Leistungsschalter, z.B. in Form von Power-MOS-FETs, enthalten, die in Stufen von 100 mA mehrere solcher Sicherungen parallelschalten. Insgesamt könne bis zu 500 mA angefordert werden. Ein externer Hub kann als Bus-Powered-Gerät nur 100 mA pro Downstrean-Port abgeben, da er insgesamt nicht mehr als 500 mA aufnehmen darf und auch noch einen gewissen Eigenbedarf hat.

2.4 USB-Transfertypen

USB-Geräte können auf vier völlig unterschiedliche Arten mit dem PC Daten austauschen.

1. Control-Transfer: Zur Steuerung der Hardware werden Control-Requests verwendet. Sie arbeiten mit hoher Priorität und mit automatischer Fehlerüberwachung. Die Übertragungsrate ist hoch, da bis zu 64 Bytes in einem Request übertragen werden können.

2. Interrupt-Transfer: Geräte, die periodisch kleine Mengen an Daten liefern, wie z.B. Mäuse und Tastaturen, verwenden den Interrupt-Transfer. Anders als der Name vermuten lässt, wird hier nicht vom Gerät ein Interrupt ausgelöst, was ja bei einem Single-Master-System auch nicht möglich wäre. Vielmehr fragt das System periodisch z.B. alle 10 ms nach neuen Daten. Typisch werden bis zu acht Bytes übertragen.
3. Bulk-Transfer: Größere Datenmengen, die eine Fehlerüberwachung benötigen, die aber nicht zeitkritisch sind, werden mit dem Bulk-Transfer übertragen. Typische Anwendungsbeispiele sind Drucker und Scanner. Die Datenrate richtet sich nach der Auslastung des USB, d.h. es wird eine niedrige Priorität verwendet.
4. Isochronous-Transfer: Große Datenmengen mit einer definierten Datenrate, wie z.B. für Soundkarten, werden isochron übertragen. Dabei wird eine bestimmte Datenrate garantiert. Eine Fehlerkorrektur findet nicht statt, da einzelne Übertragungsfehler weniger schlimm sind als Lücken in der Übertragung.

Für Anwendungen im Bereich Messen, Steuern und Regeln bietet sich besonders der Control-Transfer an. Hier wird hohe Datensicherheit mit hoher Übertragungsgeschwindigkeit gekoppelt. Es lassen sich leicht eigene Übertragungsprotokolle realisieren, die der jeweiligen Aufgabe angemessen sind.

2.5 Enumeration

Ein besonderer Vorteil des USB ist die automatische Erkennung (Plug and Play) neu angeschlossener Geräte. Das Betriebssystem muss dazu in der Lage sein, Informationen von einem Gerät abzufragen, die ihm erlauben, den passenden Treiber zu laden und das Gerät entsprechend anzusprechen. Ein neues Gerät wird dabei angemeldet (Enumeration), es erhält eine Bus-Adresse und wird durch einen speziellen Treiber unterstützt.

Die Enumeration wird völlig selbständig vom Betriebssystem ausgeführt. Weder ein Anwenderprogramm noch der Anwender muss etwas tun. Nur beim ersten Anschluss kann es sein, dass das System eine Diskette mit dem passenden Treiber verlangt. Viele Treiber sind aber bereits im System vorhanden und werden automatsich gefunden.

Der gesamte Vorgang der Enumeration beruht darauf, dass das Betriebssystem von der neu angeschlossenen Hardware bestimmte Informationen in Form von Deskriptoren abfragt. Dabei handelt es sich um genau definierte Datenblöcke von wenigen Bytes. Der PC fordert über entsprechende Control-Aufrufe diese Daten über den Endpoint 0 eines Geräts an.

Beim Anschluss eines neuen Geräts erkennt der Hub an einer hochgezogenen Datenleitung zunächst, dass ein neues Gerät vorhanden ist. Dann laufen folgende Schritte ab:

1. Der Hub informiert den Host darüber, dass ein neues Gerät angeschlossen wurde.
2. Der Host befragt den Hub, an welchem Port das Gerät angeschlossen wurde.
3. Der Host weiß nun, an welchem Port das neue Gerät angeschlossen wurde. Er gibt einen Befehl, diesen Port einzuschalten und einen Bus-Reset auszuführen.
4. Der Hub erzeugt ein Reset-Signal mit einer Länge von 10 ms. Er gibt 100 mA Versorgungsstrom für das Gerät frei. Das Gerät ist nun bereit und antwortet auf der Default-Adresse 0.
5. Bevor das USB-Gerät seine eigene Bus-Adresse erhält, kann es über die Default-Adresse Null angesprochen werden. Der Host liest die ersten Bytes des Device-Deskriptors, um festzulegen, welche Länge die Datenpakete haben können.
6. Der Host weist dem Gerät eine eigene Bus-Adresse zu.
7. Der Host liest unter der neuen Busadresse alle Konfigurations-Informationen aus dem Gerät.
8. Der Host weist dem Gerät eine der möglichen Konfigurationen zu. Das Gerät darf nun so viel Strom entnehmen, wie in seinem Configuration-Deskriptor angegeben ist. Es ist damit bereit für den Einsatz.

Jede Control-Anfrage (Request) des Host wird vom Mikrocontroller des USB-Geräts beantwortet. Dieser erkennt z.B. durch einen Hardware-Interrupt, dass Daten eingetroffen sind und im Endpoint-0-FIFO vorliegen. Die Daten werden analysiert, um die Art der Anfrage zu erkennen. Bestimmte Schlüsselbytes im Datenpaket definieren die Anforderung des Device-Deskriptors. Der Mikrocontroller liest die entsprechenden Daten aus seinem ROM und schreibt sie in das Ausgabe-FIFO, von wo sie an den Master zurückgesandt werden. Der erste abgesandte Deskriptor ist der Device-Deskriptor mit einer Länge von 18 Bytes. Der erste Zugriff des Host liest nur die ersten acht Bytes. Nach der Zuteilung der endgültigen Device-Adresse wird der ganze Deskriptor gelesen. Bei einer FIFO-Größe von 8 Bytes müssen die 18 Datenbytes auf drei Requests aufgeteilt werden.

Tabelle 2.1: Aufbau des Device-Deskriptors

Feldname	Länge, Beschreibung	Beispiel
BLength	1, Größe des Deskriptors in Byte	12h
bDescriptorType	1, Deskriptortyp (01h=Device Descriptor)	01h
BcdUSB	2, USB-Version (V. 1.0)	00h, 01h
bDeviceClass	1, Klassen-Code	00h
bDeviceSubClass	1, Subklassen-Code	00h
bDeviceProtocol	1, Protokoll-Code	00h
bMaxPacketSize0	1, Größe des EP0-FIFO	08h
IdVendor	2, Vendor-ID (04B4h=Cypress)	B4h, 04h
IdProduct	2, Produkt-ID (02=Thermometer)	02h, 00h
bcdDevice	2, Versionsnummer (V.09)	09h, 00h
iManufacturer	1, String-Index für "Hersteller"	01h
iProduct	1, String-Index für "Produkt"	02h
iSerialNumber	1, String-Index für "Seriennummer"	00h
bNumConfigurations	1, Anzahl möglicher Konfigurationen	01h

Nach dem ersten Zugriff erhält das Gerät zuerst seine endgültige Bus-Adresse. Die Bus-Adresse entspricht der Reihenfolge der angemeldeten Geräte, wobei Hubs mitgezählt werden. Da es mindestens einen Root-Hub im System gibt, ist die erste neu vergebene Adresse 2. Die neue Adresse muss in einem bestimmten Register der SIE eingetragen werden, damit im folgenden die an das Gerät gerichteten Datenpakete empfangen werden.

Aus dem Device-Deskriptor erfährt das System bereits einiges über das angeschlossene Gerät. In einigen Fällen verrät der Klassen-Code (bDeviceClass), dass das Gerät zu einer bestimmten Klasse gehört, für die das Betriebssystem selbst Treiber bereithält. So braucht der Anwender z.B. für eine USB-Maus keinen besonderen Treiber mehr. Die Maus gehört zur Klasse "Human Interface Device" (HID). Mit Hilfe dieser Informationen wird das Gerät in Betrieb genommen und im Gerätemanager eingetragen. Wenn das Gerät zu einer Klasse gehört, kann es in bDeviceSubClass und bDeviceProtocoll näher beschrieben werden. Die in diesem Buch behandelten Interfaces gehören zu keiner vordefinierten Klasse, so dass hier Nullbytes eingetragen sind.

Der Eintrag bMaxPacketSize0 beschreibt die Größe des Endpoint-0-FIFOs. Dieser Control-Endpoint muss bei jedem USB-Gerät vorhanden sein, da über ihn die Standard-Requests laufen, mit denen das System z.B. alle Deskriptoren abfragt. Der

Beim Anschluss eines neuen Geräts erkennt der Hub an einer hochgezogenen Datenleitung zunächst, dass ein neues Gerät vorhanden ist. Dann laufen folgende Schritte ab:

1. Der Hub informiert den Host darüber, dass ein neues Gerät angeschlossen wurde.
2. Der Host befragt den Hub, an welchem Port das Gerät angeschlossen wurde.
3. Der Host weiß nun, an welchem Port das neue Gerät angeschlossen wurde. Er gibt einen Befehl, diesen Port einzuschalten und einen Bus-Reset auszuführen.
4. Der Hub erzeugt ein Reset-Signal mit einer Länge von 10 ms. Er gibt 100 mA Versorgungsstrom für das Gerät frei. Das Gerät ist nun bereit und antwortet auf der Default-Adresse 0.
5. Bevor das USB-Gerät seine eigene Bus-Adresse erhält, kann es über die Default-Adresse Null angesprochen werden. Der Host liest die ersten Bytes des Device-Deskriptors, um festzulegen, welche Länge die Datenpakete haben können.
6. Der Host weist dem Gerät eine eigene Bus-Adresse zu.
7. Der Host liest unter der neuen Busadresse alle Konfigurations-Informationen aus dem Gerät.
8. Der Host weist dem Gerät eine der möglichen Konfigurationen zu. Das Gerät darf nun so viel Strom entnehmen, wie in seinem Configuration-Deskriptor angegeben ist. Es ist damit bereit für den Einsatz.

Jede Control-Anfrage (Request) des Host wird vom Mikrocontroller des USB-Geräts beantwortet. Dieser erkennt z.B. durch einen Hardware-Interrupt, dass Daten eingetroffen sind und im Endpoint-0-FIFO vorliegen. Die Daten werden analysiert, um die Art der Anfrage zu erkennen. Bestimmte Schlüsselbytes im Datenpaket definieren die Anforderung des Device-Deskriptors. Der Mikrocontroller liest die entsprechenden Daten aus seinem ROM und schreibt sie in das Ausgabe-FIFO, von wo sie an den Master zurückgesandt werden. Der erste abgesandte Deskriptor ist der Device-Deskriptor mit einer Länge von 18 Bytes. Der erste Zugriff des Host liest nur die ersten acht Bytes. Nach der Zuteilung der endgültigen Device-Adresse wird der ganze Deskriptor gelesen. Bei einer FIFO-Größe von 8 Bytes müssen die 18 Datenbytes auf drei Requests aufgeteilt werden.

Tabelle 2.1: Aufbau des Device-Deskriptors

Feldname	Länge, Beschreibung	Beispiel
BLength	1, Größe des Deskriptors in Byte	12h
bDescriptorType	1, Deskriptortyp (01h=Device Descriptor)	01h
BcdUSB	2, USB-Version (V. 1.0)	00h, 01h
bDeviceClass	1, Klassen-Code	00h
bDeviceSubClass	1, Subklassen-Code	00h
bDeviceProtocol	1, Protokoll-Code	00h
bMaxPacketSize0	1, Größe des EP0-FIFO	08h
IdVendor	2, Vendor-ID (04B4h=Cypress)	B4h, 04h
IdProduct	2, Produkt-ID (02=Thermometer)	02h, 00h
bcdDevice	2, Versionsnummer (V.09)	09h, 00h
iManufacturer	1, String-Index für "Hersteller"	01h
iProduct	1, String-Index für "Produkt"	02h
iSerialNumber	1, String-Index für "Seriennummer"	00h
bNumConfigurations	1, Anzahl möglicher Konfigurationen	01h

Nach dem ersten Zugriff erhält das Gerät zuerst seine endgültige Bus-Adresse. Die Bus-Adresse entspricht der Reihenfolge der angemeldeten Geräte, wobei Hubs mitgezählt werden. Da es mindestens einen Root-Hub im System gibt, ist die erste neu vergebene Adresse 2. Die neue Adresse muss in einem bestimmten Register der SIE eingetragen werden, damit im folgenden die an das Gerät gerichteten Datenpakete empfangen werden.

Aus dem Device-Deskriptor erfährt das System bereits einiges über das angeschlossene Gerät. In einigen Fällen verrät der Klassen-Code (bDeviceClass), dass das Gerät zu einer bestimmten Klasse gehört, für die das Betriebssystem selbst Treiber bereithält. So braucht der Anwender z.B. für eine USB-Maus keinen besonderen Treiber mehr. Die Maus gehört zur Klasse "Human Interface Device" (HID). Mit Hilfe dieser Informationen wird das Gerät in Betrieb genommen und im Gerätemanager eingetragen. Wenn das Gerät zu einer Klasse gehört, kann es in bDeviceSubClass und bDeviceProtocoll näher beschrieben werden. Die in diesem Buch behandelten Interfaces gehören zu keiner vordefinierten Klasse, so dass hier Nullbytes eingetragen sind.

Der Eintrag bMaxPacketSize0 beschreibt die Größe des Endpoint-0-FIFOs. Dieser Control-Endpoint muss bei jedem USB-Gerät vorhanden sein, da über ihn die Standard-Requests laufen, mit denen das System z.B. alle Deskriptoren abfragt. Der

Control-Endpoint nimmt eine Sonderstellung ein, da er immer beide Datenrichtungen unterstützt. Deshalb ist die Angabe der maximalen Größe eines Datenpakets im Device-Deskriptor die einzige erforderliche Information. Für Low-Speed-Geräte gilt eine Größe von 8 Bytes, während Fullspeed-Geräte größere Endpoint-0-FIFOs haben können.

Die beiden 16-Bit-Zahlen Vendor-ID und Produkt-ID identifizieren das Gerät eindeutig und erlauben die Auswahl eines entsprechenden Treibers. Die ID-Zahlen finden sich auch in der INF-Datei zum jeweiligen Treiber und ermöglichen die Zuordnung. In bcdDevice kann zusätzlich eine Versionsnummer für das Gerät eingetragen sein.

Zwei String-Indices geben die Möglichkeit, optionale Text-Deskriptoren im Gerät bereitzuhalten. Ein Hersteller kann einen Text zur näheren Beschreibung des Geräts verwenden.

Der letzte Eintrag im Device-Desktiptor bNumConfiguration gibt an, wie viele Konfigurationen das Gerät unterstützt. Ein Gerät kann mehrere Konfigurationen haben, von denen aber immer nur eine aktiviert sein kann. Ein USB-Scanner könnte z.B. zwei unterschiedliche Konfigurationen mit unterschiedlichen Stromaufnahmen haben. In bestimmten Situationen kann das Betriebssystem nur die Konfiguration mit der geringeren Stromaufnahme zulassen, während es bei nur einer Konfiguration die Enumeration des Geräts mit einer Fehlermeldung abbrechen müsste. Die meisten Geräte und die in diesem Buch beschriebenen Systeme unterstützen nur eine Konfiguration.

Die Deskriptoren besitzen eine hierarchische Struktur, die für jedes Gerät unterschiedliche Einstellungen und Eigenschaften beschreiben kann:

- Ein Gerät hat einen einzigen Device-Deskriptor.
- Es kann mehrere Konfigurationen haben.
- In jeder Konfiguration kann es mehrere Interfaces geben.
- Jedes Interfaces kann mehrere alternative Einstellungen (Alternate Settings) haben.

In einer Konfiguration können mehrere Interfaces existieren. So kann z.B. ein USB-Tastatur gleichzeitig ein Joystick-Interface und ein mausartiges Gerät enthalten. Damit hätte das USB-Gerät drei gleichzeitige Interfaces, die die vorhandenen Endpoints unter sich teilen. Nur der Control-Endpoint 0 wird von allen drei Interfaces genutzt.

Jedes Interface kann mehrere Einstellungen haben, die sich in Bezug auf die beanspruchte Bandbreite unterscheiden. Besonders bei der Verwendung isochroner

Endpoints wird viel Bandbreite reserviert. Es kann daher mehrere Alternate Settings mit unterschiedlichen Endpoint-FIFO-Größen geben. Alternate Setting 0 sollte nur den Endpoint 0 verwenden, damit die Möglichkeit besteht, das Gerät in den unkonfigurierten Zustand zu versetzen, in dem es keine Bandbreite beansprucht.

Außer dem Device-Deskriptor wird noch der Configuration-Deskriptor, einer oder mehrere Interface-Deskriptoren, Endpoint-Deskriptoren und optional String-Deskriptoren abgefragt. Nach und nach erfährt das System immer mehr über das Gerät, z.B. wie viele Endpoints mit welcher FIFO-Größe es hat, ob es mehrere Konfigurationen unterstützt, mehrere Interfaces hat und wie viele Alternate-Settings jedes Interface besitzt.

Bei der Abfrage des Configuration-Dekriptors durch das Betriebssystem werden mehrere Deskriptoren zusammengefasst. Zur Konfiguration gehören auch noch die Interface-Deskriptoren und alle zugehörigen Endpoint-Deskriptoren. Deshalb gibt der Konfigurations-Dekriptor in wTotalLength die Gesamtlänge aller zusammengehörenden Deskriptoren an.

Tabelle 2.2: Aufbau des Configuration-Deskriptors

Feldname	Länge, Beschreibung	Beispiel
bLength	1, Größe des Deskriptors in Byte	09h
bDescriptorType	1, Deskriptortyp (02h=Configuration Descriptor)	02h
wTotalLength	Länge aller Deskriptoren dieser Konfguration	22h, 00h
bNumInterfaces	Anzahl der Interfaces	01h
bConfigurationValue	Nummer dieser Konfiguration	01h
iConfiguration	String-Index	00h
bmAttributes	Attribute der Konfiguration	04h
MaxPower	Stromaufnahme, Einheit 2 mA	32h

Der Konfigurations-Deskriptor beschreibt die Anzahl der Interfaces und die aktuelle Konfiguration sowie eventuell vorhandene String-Deskriptoren. In bmAtrigutes kann angegeben werden, ob das Gerät vom Bus mit Strom versorgt wird und ob es eine Remote-Wakeup-Funktion unterstützt, also von sich aus den Bus aufwecken kann. In MaxPower wird der Strombedarf vom Bus angegeben. Die Angabe 50 (=32h) bedeutet 100 mA und steht für die ohne weiteres erlaubte Stromentnahme. Dieser Strom kann auch von einem Hub im Self-Powered-Modus geliefert werden.

Tabelle 2.3: Aufbau des Interface-Deskriptors

Feldname	Länge, Beschreibung	Beispiel
bLength	1, Größe des Deskriptors in Byte	09h
bDescriptorType	1, Deskriptortyp (04h=Interface Descriptor)	04h
bInterfaceNumber	Interface-Nummer	00h
bAlternateSetting	Alternative Setting	00h
bNumEndpoints	Anzahl der Endpoints ohne EP0	00h
bInterfaceClass	Eventuelle Interface-Klasse	00h
bInterfaceSubClass	Eventuelle Interface-Subklasse	00h
bInterfaceProtocol	Eventueller Protokoll-Code	00h
iInterface	String-Index	00h

Es kann mehrere Interface-Deskriptoren mit jeweils einer Alternate Setting geben. Die wichtigste Information zu jeder Einstellung ist die Anzahl der Endpoints, wobei zu jedem Endpoint ein Endpoint-Deskriptor gehört. Falls das Gerät zu einer Geräte-Klasse gehört, die bereits im Device-Deskriptor angegeben wurde, können die entsprechenden Informationen hier erneut gegeben werden.

Tabelle 2.4: Aufbau des Endpoint-Deskriptors

Feldname	Länge, Beschreibung	Beispiel
bLength	1, Größe des Deskriptors in Byte	07h
bDescriptorType	1, Deskriptortyp (05h=Endpoint-Deskriptor)	05h
bEndpointAddress	Endpoint-Adresse (EP2, OUT)	02h
bmAttributes	Transfer-Typ (2=Bulk-Transfer)	02h
wMaxPacketSize	FIFO-Größe	40h, 00h
bInterval	Eventuelles Polling-Intervall	00h

Die Endpoint-Adresse enthält im höchstwertigen Bit die Information über die Datenrichtung. Die Angabe bEndpointAddress=82h bedeutet Endpoint-2-IN. Für jeden Endpoint ist der Transfer-Typ festgelegt. 0 steht für Control-Transfer, 1 für isochronen Verkehr, 2 für Bulk-Transfer und 3 für Interrupt-Transfer. Wichtig ist auch die FIFO-Größe in wMaxPacketSize. Nur für Interrupt- und isochronen Transfer muss eine Polling-Intervall angegeben werden.

Bei der Enumeration werden alle wichtigen Informationen des Geräts abgefragt. Das Betriebssystem überprüft dabei, ob die angeforderten Recourcen wie Treiber, Stromverbrauch und Bus-Bandbreite vorhanden sind. Nach einer erfolgreichen Enumeration kann man sofort Control-Requests über den Endpoint 0 verwenden. Die anderen Endpoints und Betriebsarten müssen erst durch die Zuweisung einer Alternate-Setting > 0 eingeschaltet werden.

Der gesamte Vorgang der Enumeration ist sehr komplex und soll in diesem Buch nicht vollständig beschrieben werden. Die verwendeten Geräte und Mikroprozessoren bieten hier bereits fertige Lösungen oder Beispielsoftware. Der Anwender braucht sich zumindest bei seinen ersten Schritten nicht mit allen Einzelheiten zu belasten. Es reicht zunächst, ein bereits enumeriertes Gerät zu betrachten und den Datenaustausch mit der Anwendung zu planen.

Vollständige Informationen zur Enumeration und zu allen definierten Deskriptoren finden sich in der Literatur und in den Spezifikationen der USB-Organisation, die als PDF-File auf der CD vorliegen. Hilfreich ist auch die Analyse der vorliegenden Referenz-Designs von Cypress.

2.6 Treiber-Aufrufe

Die Funktion eines Treibers kann allgemein als Bindeglied zwischen der Hardware und der Software eines Systems beschrieben werden. Unter Windows 98 darf kein Anwenderprogramm direkt auf die Hardware zugreifen. Vielmehr muss ein Treiber aufgerufen werden, der die Hardware direkt oder über tiefer liegende Treiberschichten anspricht. Programme arbeiten im User Mode und haben keine Zugriffsrechte auf Hardware. Treiber arbeiten im Kernel Mode mit höheren Rechten. Ein Treiber wird wie ein virtuelles Gerät betrachtet, d.h. ein Anwenderprogramm verwendet einen Treiber, als wäre es das Gerät selbst.

Alle USB-Treiber basieren auf dem Win32-Driver-Model, das bereits in Windows NT verwendet wurde und in Windows 98 adaptiert wurde. Alle Treiberaufrufe laufen über den I/O-Manager. Verschiedene Anwenderprogramme senden quasi gleichzeitig Anforderungspakete an den I/O-Manager, der sie dann an die einzelnen Treiber verteilt. Im Fall des USB laufen mehrere Requests von verschiedenen Programmen über verschiedene Gerätetreiber zu einem allgemeinen USB-Treiber, der letztlich alle angeschlossenen Geräte bedient. Jeder Gerätetreiber steuert den tiefer liegenden USB-Treiber des Betriebssystems an. Dieser fügt die einzelnen Requests zu Gesamtpaketen zusammen und steuert damit die Hardware an.

Alle Requests werden in Form von Paketen (I/O-Request-Packet, IRP) versandt und zwischen der einzelnen Treiberschichten weitergereicht. Auf der Ebene des An-

wenderprogramms benötigt man nur fünf Windows-Funktionen, um auf alle denkbaren Geräte zuzugreifen. Geräte werden wie Dateien behandelt und mit CreateFile() geöffnet. Schreibzugriffe verwenden z.B. beim Bulk-Transfer im Regelfall WriteFile(), Lesezugriffe ReadFile(). Mit CloseHandle() schließt man den Zugriff. Für alle Zugriffe, die nicht in das Muster einer Datei passen, gibt es den Zugriff über DeviceIoControl(). So werden spezielle Parameter und Steuerdaten zwischen Anwenderprogramm und Gerät ausgetauscht. Ein Gerät kann also z.B. bei der Initialisierung mit bestimmten Einstellungen versehen werden.

Tabelle 2.5: Windows-IO-Funktionen

CreateFile()	Öffnen einer Datei oder eines Geräts
CloseHandle()	Schließen einer Datei oder eines Geräts
ReadFile()	Lesen bzw. Empfangen von Daten
WriteFile()	Schreiben bzw. Senden von Daten
DeviceIoControl()	Ausführen spezieller Treiberfunktionen

Zur Verdeutlichung des grundsätzlichen Zugriffs auf Gerätetreiber soll hier ein einfaches Beispiel für die serielle Schnittstelle gezeigt werden. Ähnlich wie beim USB sorgt ein Systemtreiber dafür, dass die eigentliche Hardware angesteuert wird.

Die Schnittstelle COM2 wird mit CreateFile geöffnet, um dann ein Byte mit WriteFile zu senden. CloseHandle schließt den Kommunikationskanal schließlich wieder. Der Name des Treibers ist hier „COM2", er repräsentiert also die Hardware-Schnittstelle „COM2". Entsprechend steht der Name „\\.\Thermometer_0" für ein spezielles USB-Gerät, das in den folgenden Kapiteln verwendet wird.

Beim Öffnen wird ein Handle verwendet, also eine Zahl vom Typ Integer, die zur eindeutigen Zuordnung des verwendeten IO-Kanals dient. Das Handle wird vom System entsprechend der Reihenfolge von CreateFile-Aufrufen vergeben. Da ein Aufruf wie WriteFile für sehr unterschiedliche Aktionen und Geräte verwendet werden kann, unterscheidet das Betriebssystem diese mit Hilfe des Handles. Entsprechend gibt CloseHandle das entsprechende Gerät wieder frei.

```
Handle:=CreateFile(PChar('COM2'),GENERIC_WRITE,0,NIL,
  OPEN_EXISTING,0,0);
 Byt := 85;
 WriteFile(Handle,Byt,1,Count,NIL);
 CloseHandle(Handle);
```

Listing 2.1: Senden eines Bytes über COM2 in Delphi

Dieses Beispiel zeigt zwar den grundsätzlichen Zugriff auf die Schnittstelle, es hat jedoch noch keinen praktischen Nutzen, weil Schnittstellenparameter wie die Baudrate oder die Anzahl der Datenbits nicht explizit festgelegt wurden.

2.7 CreateFile und CloseFile

Die Funktion CreateFile hat zahlreiche Parameter, die hier im einzelnen erläutert werden sollen. Weitere Informationen findet man in der Hilfe-Datei WIN32.HLP. Die Deklaration erfolgt hier in C-Schreibweise. In Delphi 4 ist sie in der Unit WINDOWS.PAS entsprechend umgesetzt.

```
HANDLE CreateFile(

  LPCTSTR lpFileName,                          // pointer to name of the file
  DWORD dwDesiredAccess,                       // access (read-write) mode
  DWORD dwShareMode,                           // share mode
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,  // pointer to security
                                               // attributes
  DWORD dwCreationDistribution,                // how to create
  DWORD dwFlagsAndAttributes,                  // file attributes
  HANDLE hTemplateFile                         // handle to file with
                                               // attributes to copy
);
```

lpFileName	Zeiger auf einen nullterminierten String mit dem Namen des Geräts oder der Datei.
dwDesiredAccess	Konstante zur Beschreibung des Schreib- oder Lesezugriffs 0: kein Zugriff GENERIC_READ= $80000000:Lesezugriff GENERIC_WRITE=$40000000:Schreibzugriff
dwShareMode	Gibt an, ob eine Datei von mehreren Anwendungen geteilt werden kann
lpSecurityAttributes	Zeiger auf eine Struktur von Sicherheitsattributen, für Schnittstellen nicht unterstützt, daher immer NIL
dwCreationDistribution	Gibt für Dateien an, ob sie neu angelegt werden sollen oder nur geöffnet werden sollen, wenn sie bereits existieren. Für Schnittstellen ist die einzige mögliche Einstellung OPEN_EXISTING = 3.
dwFlagsAndAttributes	Übergibt Attribute für Dateien, für Schnittstellen meist 0 oder FILE_FLAG_OVERLAPPED
hTemplateFile	Übergibt ein Handle auf eine Vorlagedatei mit erweiterten Attibuten. Für Schnittstellen immer 0 .

CreateFile gibt bei Erfolg ein gültiges Handle zurück. Falls das Öffnen nicht möglich war, weil die Schnittstelle nicht existiert oder bereits belegt war, ist der Rückgabewert die Konstante:

```
INVALID_HANDLE_VALUE =-1;
```

Ein grundlegender Unterschied besteht in der Verwendung eines Geräts im Overlapped-Modus (auch als asynchron bezeichnet) oder im Nonoverlapped-Modus (auch: synchron). Im Normalfall verwendet man Nonoverlapped, d.h. ein Prozess wird angehalten, bis eine Schreib- oder Leseaktion der Schnittstelle abgeschlossen ist. Eine Prozedur, die einen längeren Datenblock sendet, wird also erst dann wieder verlassen, wenn alle Zeichen vollständig übertragen sind. Andere Programme laufen aber ungestört weiter, da das Multitasking nicht beeinträchtigt wird. Nur der eigene Thread (Prozess) wird angehalten. Insbesondere beim Empfang von Daten kann ein Programm „hängen", wenn weniger als die angeforderten Bytes tatsächlich eintreffen.

Im Overlapped-Modus läuft ein Thread weiter, auch während eine Übertragung zu oder von einem Gerät noch läuft. Übertragungszeiten können gleichzeitig für andere Vorgänge benutzt werden. Dazu ist aber eine Synchronisierung in Form von Nachrichten über den Stand einer Schnittstellenaktion erforderlich.

Jedes geöffnete Gerät muss nach der Verwendung wieder geschlossen werden, um es für andere Anwendungen freizugeben. Dazu dient die Funktion CloseHandle.

```
BOOL CloseHandle(
  HANDLE hObject  // handle to object to close
   );
```

2.8 WriteFile und ReadFile

Der eigentliche Schreibzugriff auf Geräte wie die serielle Schnittstelle oder den USB erfolgt unter Win32 mit WriteFile(). Der Zugriff entspricht dem Schreiben in eine Diskettendatei.

```
BOOL WriteFile(
    HANDLE hFile,                    // handle to file to write to
    LPCVOID lpBuffer,                // pointer to data to write to file
    DWORD nNumberOfBytesToWrite,     // number of bytes to write
    LPDWORD lpNumberOfBytesWritten,  // pointer to number of bytes written
    LPOVERLAPPED lpOverlapped        // pointer to structure needed for
                                     // overlapped I/O
   );
```

hFile	Handle auf das geöffnete File oder Gerät
lpBuffer	Zeiger auf einen Datenpuffer
nNumberOfBytesToWrite	Anzahl der zu sendenden Bytes
lpNumberOfBytesWritten	Zeiger auf eine Variable mit der Anzahl der tatsächlich geschriebenen Bytes
lpOverlapped	Zeiger auf eine Overlapped- Struktur oder NIL für Nonoverlapped

Für die Leserichtung lautet die Funktion ReadFile.

```
BOOL ReadFile(
    HANDLE hFile,                  // handle of file to read
    LPVOID lpBuffer,               // address of buffer that receives data
    DWORD nNumberOfBytesToRead,    // number of bytes to read
    LPDWORD lpNumberOfBytesRead,   // address of number of bytes read
    LPOVERLAPPED lpOverlapped      // address of structure for data
   );
```

hFile	Handle auf das geöffnete File oder Gerät
lpBuffer	Zeiger auf einen Datenpuffer für den Empfang
nNumberOfBytesToRead	Anzahl der zu empfangenden Bytes
lpNumberOfBytesRead	Zeiger auf eine Variable mit der Anzahl der tatsächlich empfangenen Bytes
lpOverlapped	Zeiger auf eine Overlapped-Struktur oder NIL für Nonoverlapped

2.9 DeviceIoControl

Spezielle Steuerzugriffe auf einen Treiber erfolgen mit DeviceIoControl. Diese Funktion führt immer einen Datentransfer in beiden Richtungen durch. Solche Zugriffe eignen sich auch für Mess- und Steuerungsaufgaben. Meist wird hier für den USB der Control-Transfer verwendet. Eine intern festgelegte Treiber-Funktionsnummer steuert die Auswahl einer bereitgestellten Spezialfunktion. Der Anwender übergibt diese Funktionsnummer (I/O-Control-Code) und eventuell zusätzliche Daten. Er erhält entsprechend der aufgerufenen Steuerfunktion Daten zurück.

```
BOOL DeviceIoControl(
    HANDLE hDevice,              // handle to device of interest
    DWORD dwIoControlCode,       // control code of operation to perform
    LPVOID lpInBuffer,           // pointer to buffer to supply input data
    DWORD nInBufferSize,         // size of input buffer
    LPVOID lpOutBuffer,          // pointer to buffer to receive output data
    DWORD nOutBufferSize,        // size of output buffer
    LPDWORD lpBytesReturned,     // pointer to variable to receive output byte
                                 // count
    LPOVERLAPPED lpOverlapped    // pointer to overlapped structure for
                                 // asynchronous operation
);
```

HDevice	Handle auf das geöffnete File oder Gerät
DwIoControlCode	Control-Code für die Gerätefunktion
LpInBuffer	Zeiger auf einen Input-Datenpuffer
NInBufferSize	Größe des Input-Datenpuffers
LpOutBuffer	Zeiger auf einen Output-Datenpuffer
NOutBufferSize	Größe des Output-Datenpuffers
LpBytesReturned	Zeiger auf eine Variable mit der Anzahl der tatsächlich zurückgelieferten Bytes
LpOverlapped	Zeiger auf eine Overlapped-Struktur oder NIL für Nonoverlapped

Eine typische Anwendung für DeviceIoControl () ist das Auslesen von Speicherstellen oder Registern eines USB-Controllers. Der Treiber muss dazu eine entsprechende Funktion bereitstellen, die mit einem I/O-Control-Code aufgerufen wird. Das Anwenderprogramm übergibt im Input-Puffer die Adresse und eventuelle andere Parameter und erhält im Output-Puffer die gelesenen Speicherinhalte zurück.

Für die Festlegung des I/O-Control-Code, einer im Treiber festgelegten 32-Bit-Konstanten, gibt es Konventionen, an die sich ein Treiberhersteller halten sollte. Man findet diese Definitionen z.B. in der Datei Devioctl.h in der Win98-DDK.

```
#define CTL_CODE( DeviceType, Function, Method, Access )
  (((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method))
```

DeviceType ist eine den einzelnen USB-Geräteklassen zugeordnete Konstante. Besonders wichtig ist die Klasse FILE_DEVICE_UNKNOWN, mit der man nicht vordefinierte Spezialanwendungen realisiert.

```
#define DEVICE_TYPE ULONG
#define FILE_DEVICE_BEEP                    0x00000001
#define FILE_DEVICE_CD_ROM                  0x00000002
#define FILE_DEVICE_CD_ROM_FILE_SYSTEM      0x00000003
#define FILE_DEVICE_CONTROLLER              0x00000004
#define FILE_DEVICE_DATALINK                0x00000005
#define FILE_DEVICE_DFS                     0x00000006
#define FILE_DEVICE_DISK                    0x00000007
#define FILE_DEVICE_DISK_FILE_SYSTEM        0x00000008
#define FILE_DEVICE_FILE_SYSTEM             0x00000009
#define FILE_DEVICE_INPORT_PORT             0x0000000a
#define FILE_DEVICE_KEYBOARD                0x0000000b
#define FILE_DEVICE_MAILSLOT                0x0000000c
#define FILE_DEVICE_MIDI_IN                 0x0000000d
#define FILE_DEVICE_MIDI_OUT                0x0000000e
#define FILE_DEVICE_MOUSE                   0x0000000f
#define FILE_DEVICE_MULTI_UNC_PROVIDER      0x00000010
#define FILE_DEVICE_NAMED_PIPE              0x00000011
#define FILE_DEVICE_NETWORK                 0x00000012
#define FILE_DEVICE_NETWORK_BROWSER         0x00000013
#define FILE_DEVICE_NETWORK_FILE_SYSTEM     0x00000014
#define FILE_DEVICE_NULL                    0x00000015
#define FILE_DEVICE_PARALLEL_PORT           0x00000016
#define FILE_DEVICE_PHYSICAL_NETCARD        0x00000017
#define FILE_DEVICE_PRINTER                 0x00000018
#define FILE_DEVICE_SCANNER                 0x00000019
#define FILE_DEVICE_SERIAL_MOUSE_PORT       0x0000001a
#define FILE_DEVICE_SERIAL_PORT             0x0000001b
#define FILE_DEVICE_SCREEN                  0x0000001c
#define FILE_DEVICE_SOUND                   0x0000001d
#define FILE_DEVICE_STREAMS                 0x0000001e
#define FILE_DEVICE_TAPE                    0x0000001f
#define FILE_DEVICE_TAPE_FILE_SYSTEM        0x00000020
#define FILE_DEVICE_TRANSPORT               0x00000021
#define FILE_DEVICE_UNKNOWN                 0x00000022
#define FILE_DEVICE_VIDEO                   0x00000023
#define FILE_DEVICE_VIRTUAL_DISK            0x00000024
#define FILE_DEVICE_WAVE_IN                 0x00000025
#define FILE_DEVICE_WAVE_OUT                0x00000026
#define FILE_DEVICE_8042_PORT               0x00000027
#define FILE_DEVICE_NETWORK_REDIRECTOR      0x00000028
#define FILE_DEVICE_BATTERY                 0x00000029
#define FILE_DEVICE_BUS_EXTENDER            0x0000002a
#define FILE_DEVICE_MODEM                   0x0000002b
```

Access beschreibt den Datenzugriff als Schreib- oder Lesezugriff:

```
#define FILE_ANY_ACCESS              0
#define FILE_READ_ACCESS          ( 0x0001 )    // file & pipe
#define FILE_WRITE_ACCESS         ( 0x0002 )    // file & pipe
```

Function ist die vom Hersteller des Treibers festgelegte Funktionsnummer. Dabei sind die Funktionsnummern 0...2047 für Microsoft reserviert, während Funktionsnummern von 2048 bis 4095 von anderen Herstellern verwendet werden können. Die Firma Anchor-Chips verwendet z.B. für den EZUSBSYS-Treiber Funktionscodes ab 800h=2048. Die einzelnen Funktionen sind mit aufsteigenden Konstanten festgelegt.

Method beschreibt die Methode, mit der Control-Aufrufe durchgeführt werden. Am meisten wird METHOD_BUFFERED verwendet. Die Daten werden in vom Anwenderprogramm bereitgestellten Puffern übergeben.

```
#define METHOD_BUFFERED      0
#define METHOD_IN_DIRECT     1
#define METHOD_OUT_DIRECT    2
#define METHOD_NEITHER       3
```

Zur Verdeutlichung der Vorgänge bei einem Control-Transfer soll hier ein konstruiertes Beispiel vorgestellt werden. Ein USB-Gerät habe mehrere Ports, die über den USB ausgelesen werden können. Für die Funktion Port-Auslesen sei im Treiber ein I/O-Control-Code festgelegt. Zusätzlich erwartet der Treiber eine Portnummer, die im Input-Buffer übergeben werden soll.

Der Aufruf aus dem Anwenderprogramm verwendet also DeviceIoControl () mit der für die Funktion Read-Port festgelegten Funktionsnummer. Zusätzlich wird im Input-Puffer die Portnummer übergeben. Die Länge der übergebenen Daten beträgt ein Byte. Außerdem muss ein Output-Puffer mit einer Länge von einem Byte bereitgestellt werden, da bekannt ist, dass der Treiber ein Byte zurückliefert.

Der Treiber wertet zunächst die Funtionsnummer aus, um festzustellen, welche Aktion das Anwenderprogramm verwendet. Er erkennt die Funktion Port-Lesen und baut ein passendes Request-Paket zur Übermittlung an die Hardware auf. Dieses muss eine Funktionsnummer Port-Lesen für den USB-Controller und die gewünschte Portnummer enthalten. Die Funktionsnummer für die Steuerung der Hardware ist nicht identisch mit der für den Treiber. Man verwendet hier z.B. nur ein Byte.

Das Request-Paket wird nun an den tiefer liegenden USB-Treiber des Systems geschickt. Dieser ordnet es zusammen mit anderen Requests zu einem Datenrahmen zusammen und überträgt diesen an den Host-Controller. Von dort laufen die Daten über einen oder mehrere Hubs zum USB-Gerät, wo sie von der SIE empfangen werden.

Der angeschlossene Mikrocontroller wird mit einem Interrupt vom Eintreffen einer Nachricht unterrichtet. Er analysiert die im Empfangs-FIFO vorliegenden Daten und erkennt die gewünschte Funktion Port-Lesen und die Portnummer. Er liest

den entsprechenden Port und schreibt das Byte in das Sende-FIFO. Dann erteilt er der SIE den Auftrag zum Rücksenden der Daten.

Der gelesene Portzustand, also die Nutzdaten von einem Byte, wird über den Bus zurückgesandt und gelangt durch alle Hubs bis zum Host-Controller. Dieser empfängt die Daten und gibt sie über den USB-Treiber an den Gerätetreiber zurück. Der Gerätetreiber schreibt das empfangene Byte in den von der Anwendersoftware bereitgestellten Output-Puffer und teilt dem aufrufenden Programm mit, dass ein Byte übergeben wurde.

Damit schließt sich die Übertragungskette. Da der hin- und herlaufende Datenverkehr über den USB mit einer Fehlerabfrage auf mehrere Datenrahmen verteilt wird, dauert die Abwicklung insgesamt drei Millisekunden. Für eine fortgesetzte Abfrage eines Portzustands erreicht man einen Taktzyklus von 4 ms. Das bedeutet eine relativ geringe effektive Übertragungsrate. Um den USB effektiver zu nutzen, sollten mehrere Bytes pro Frame transportiert werden. Beim Auslesen ganzer Datenbereiche erreicht man bis zu 64 Bytes in 4 ms. Das entspricht 16 Kilobytes pro Sekunde oder 128 Kilobit pro Sekunde.

3 USB-Standardgeräte

Der erste Kontakt mit dem USB kommt für die meisten Anwender über Standard-Peripheriegeräte des PC. Es lohnt sich, die Installation und den Betrieb der Geräte genauer zu analysieren, da sich hier vieles für die Entwicklung spezieller Geräte ableiten lässt. Außerdem können auch einige speziellere Aufgaben aus dem Bereich Messen, Steuern und Regeln mit Standardgeräten durchgeführt werden.

3.1 Installation einer USB-Maus

Die Maus eines PCs kann an der RS232-Schnittstelle, an einem PS2-Anschluss oder am USB angeschlossen sein. USB-Mäuse gehören inzwischen schon zum Standard. Es ist interessant, die Installation einer solchen Maus genauer zu betrachten.

Eine USB-Maus gehört zur Gruppe der Human Interface Devices (HID). Sie wird damit vom Betriebssystem Windows 98 durch besondere Treiber unterstützt, um die sich der Anwender nie kümmern muss. Im laufenden Betrieb werden Daten der Maus in regelmäßigen Abständen über Interrupt-Requests abgefragt. Für den Anwender ist der Anschluss der Maus nicht mehr wichtig, die USB-Maus verhält sich also genauso wie eine Maus an der RS232.

Beim ersten Anschluss der USB-Maus erkennt das Betriebssystem eine neue Hardware. Der Hardware-Assistent von Windows-98 meldet sich und gibt eine Meldung über die Art der gefundenen Hardware (vgl. *Abb. 3.1*).

Abb. 3.1: Automatische Suche nach einem Treiber

Bei der Suche nach einem passenden Treiber wird zunächst im System selbst geforscht. Dort findet sich bereits die Datei HIDDEV.INF, so dass der Benutzer keinen Treiber bereitstellen muss (vgl. *Abb. 3.2*).

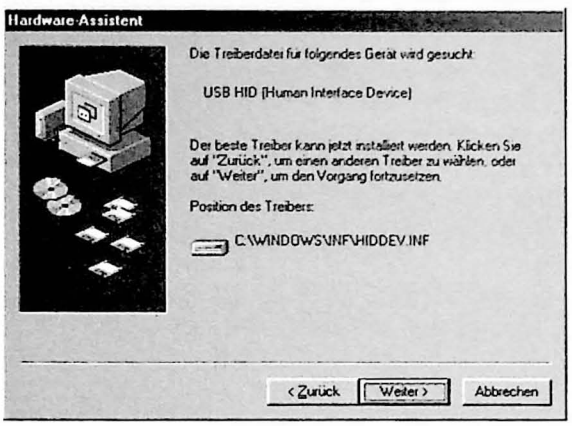


Abb. 3.2: Die gefundene INF-Datei des Treibers

Bei der ersten Enumeration der USB-Maus wurden bestimmte Informationen in Form von Deskriptoren ausgelesen. Daraus wurde die Zuordnung zur HID-Klasse und zum Klassentyp „Maus" erkannt. Die zugehörige INF-Datei enthält Einträge über die Hardware und den passenden Treiber.

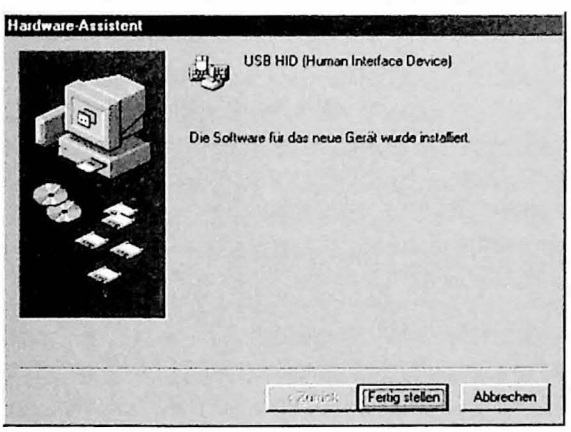


Abb. 3.3: Der Abschluss der Installation

Nach der erfolgreichen Installation kann die neue Maus wie gewohnt verwendet werden. Behält man die schon vorhandene Maus bei, kann man den PC nun mit zwei Mäusen bedienen. Die USB-Maus kann aber von nun an problemlos bei Bedarf angeschlossen und wieder getrennt werden. Bei jedem neuen Anschluss er-

scheint nur kurz die Windows-Sanduhr auf dem Schirm und zeigt das neue Laden des eingetragenen Treibers. Alle erforderlichen Informationen befinden sich nun in der Windows-Registry.

Jedes neue Gerät benötigt einen Treiber und eine zugehörige INF-Datei. Listing 3.1 zeigt Auszüge aus der Datei HIDDEV.INF. Während Standard-Klassen bereits vom Betriebssystem unterstützt werden, müssen spezielle Geräte mit den entsprechenden Dateien auf einer Diskette geliefert werden.

```
; Localized 05/01/1998 09:38  (GMT) 7:3.0.406 A hiddev.inf

;
; HIDDEV.INF
;
; - Installs default support for HID devices according to compatible id
;
; Copyright (c) 1998 Microsoft Corporation
;

....

 [Strings]
UnknownMfg = "(Standardgerät)"
USB\Class_03 = "USB HID (Human Interface Device)"
HIDClassDescription = "HID (Human Interface Devices)"
HID_DEVICE_SYSTEM_CONTROL = "HID-kompatibles Systemsteuergerät"
HID_DEVICE_SYSTEM_CONSUMER = "HID-kompatibles Steuerungsgerät"
HID_DEVICE = "HID-kompatibles Gerät"
```

Listing 3.1: Auszüge aus der INF-Datei HIDDEF.INF

3.2 Eine USB-Soundkarte

Soundkarten werden vielfach bereits zu Messzwecken eingesetzt. Der Vorteil einer USB-Soundkarte ist der leichtere Zugang der Anschlüsse und die weniger gestörte Umgebung außerhalb des PC. Es gibt einfache USB-Ausgabegeräte zum Anschluss an Kopfhörer oder Aktivboxen. Aufwendigere Geräte besitzen auch Eingänge für Sound-Aufnahmen.

Hier wurde ein USB-Sound-Ausgabegerät vom Typ hama CS-495 verwendet. Das Gerät installiert sich beim ersten Plug-In als USB-Soundgerät. Wenn schon eine Soundkarte im PC vorhanden ist, muss das neue Gerät im Gerätemanager als bevorzugtes Ausgabegerät angegeben werden. Sound-Ausgaben laufen dann immer über den USB (vgl. *Abb. 3.4*), während Aufnahmen weiterhin über die interne Soundkarte erfolgen.

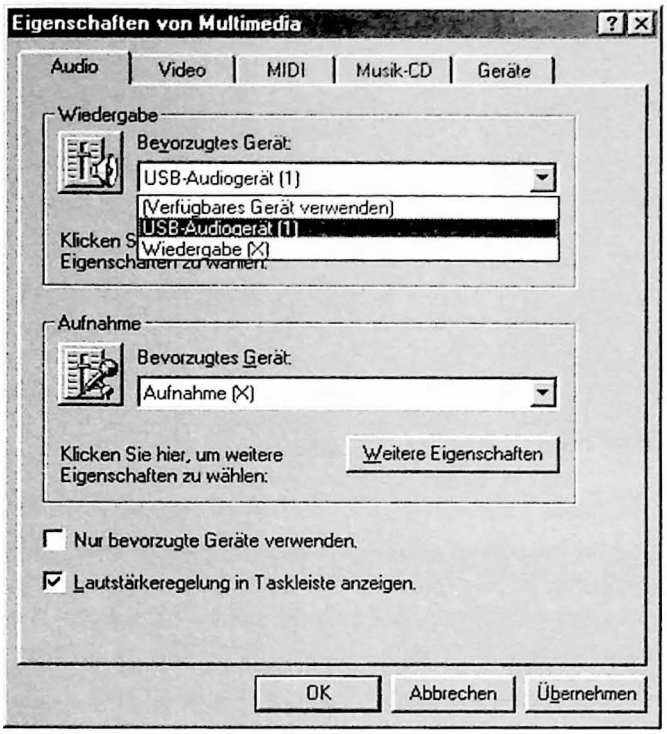


Abb. 3.4: Das neue Multimedia-Gerät

Das Gerät verwendet intern den Philips-Soundchip UDA1321PS/N101. Erste Versuche zeigen, dass das Gerät zwischen 20 Hz und 20 kHz einen sehr ausgeglichenen Frequenzgang besitzt. Es eignet sich daher für den Einsatz als Tongenerator für allgemeine Messzwecke. Der Aufbau als externes Gerät (vgl. *Abb. 3.5*) ermöglicht bei Bedarf das einfache Überbrücken der Ausgangs-Koppelkondensatoren zur weiteren Herabsetzung der unteren Grenzfrequenz.

Das Datenblatt des UDA1321 gibt einen guten Einblick in den grundsätzlichen Aufbau des Geräts. Der USB-Chip enthält einen vollständigen Mikrocontroller zur Enumeration des Geräts und zur Durchführung des isochronen Datenverkehrs vom PC zur Sound-Ausgabe. Über ein externes EEPROM kann der Hersteller die eigenen IDs verwenden. Ist kein EEPROM angeschlossen, meldet sich das Gerät mit dem Herstellernamen Philips.

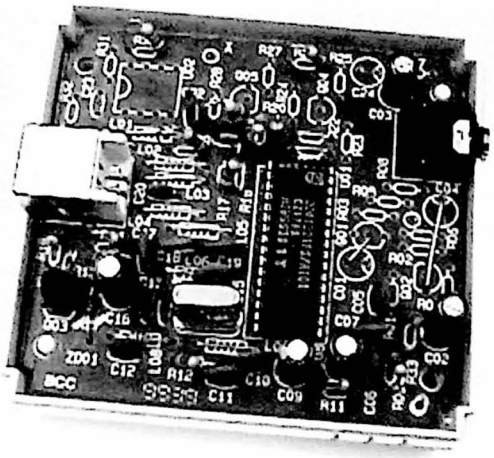


Abb. 3.5: Das Innere des USB-Soundgeräts

Das IC wird in verschiedenen Gehäuseformen angeboten. Zum Teil werden Leitungen für den Anschluß eines externen ROMs herausgeführt, so dass der Hersteller eine eigene Firmware einsetzen kann. Im vorliegenden Gerät wird der Chip mit der internen Firmware von Philips betrieben.

Die Sound-Ausgabe erfolgt in Form von Wave-Daten. Es wird also mit der vorgegebenen Sample-Rate von z.B. 44100 Samples pro Sekunde jeweils zwei Bytes pro Kanal übertragen. Dabei darf keine Lücke im Datenstrom auftreten. Die erforderliche Datenrate wird über den isochronen Transfer des USB erreicht. Die Bus-Belastung ist wesentlich höher als z.B. bei einem HID-Gerät wie der Maus, wo nur geringe Datenmengen anfallen. Die Belegung der Bus-Bandbreite lässt sich sehr einfach mit einem Oszilloskop untersuchen. Man misst dabei die Signale an der USB-Datenleitung D+ oder D-. Es ist deutlich der Millisekunden-Rahmen zu erkennen. In jedem Rahmen ist ein gewisses Zeitfenster belegt. Das Sound-Ausgabeberät belegt z.B. 15% der verfügbaren Bandbreite (vgl. Abb.3.7). Die direkte Messung mit dem Oszilloskop ist immer dann sehr hilfreich, wenn Bus-Aktivitäten festgestellt werden sollen.

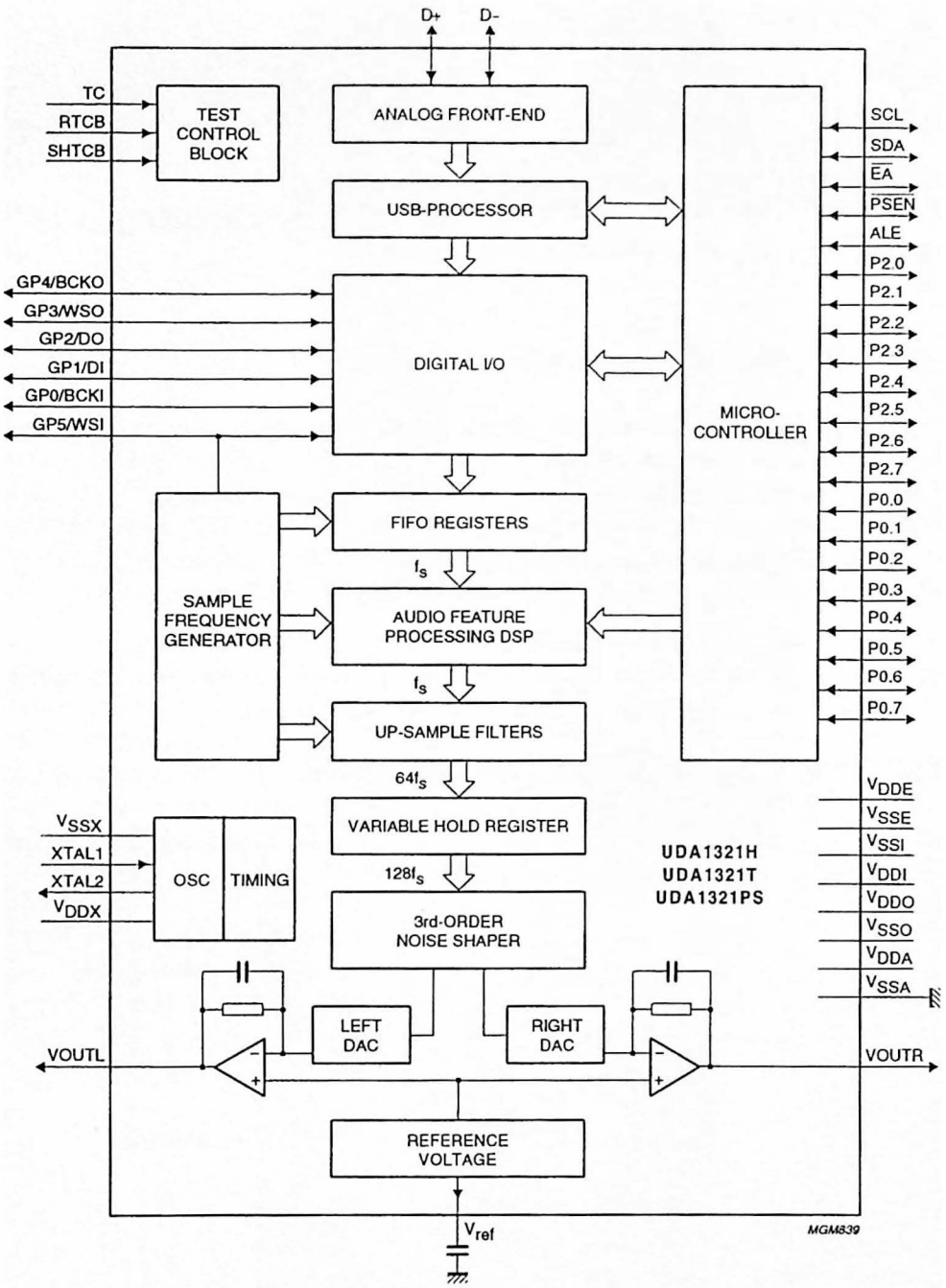


Abb. 3.6: Blockschaltbild des UDA1321 (Philips)

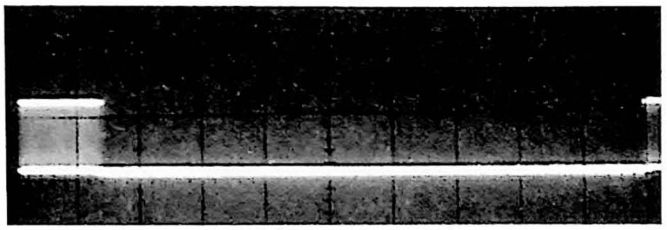


Abb. 3.7: Belegung der USB-Bandbreite bei Soundausgabe

3.3 Der Signalgenerator AUDIO-Wave

Das Programm Audio-Wave der Firma Abacom (www.abacom-online.de) realisiert einen allgemeinen Sinusgenerator mit einer beliebigen Soundkarte, also auch mit einer installierten USB-Soundausgabe. Es können Frequenzen bis zu 20 kHz erzeugt werden. Die Eingabe der Frequenz erfolgt nummerisch oder über Drucktasten. Die Ausgangspegel lassen sich über entsprechende Knöpfe beeinflussen. Zwischen beiden Ausgabekanälen kann eine Phasenverschiebung eingestellt werden.

Außer konstanten Signalen können auch modulierte Signale oder Frequenz-Verläufe gewählt werden. Damit lassen sich z.B. Filterkurven und Frequenzgänge untersuchen. Auch ein weißes Rauschen kann erzeugt werden.

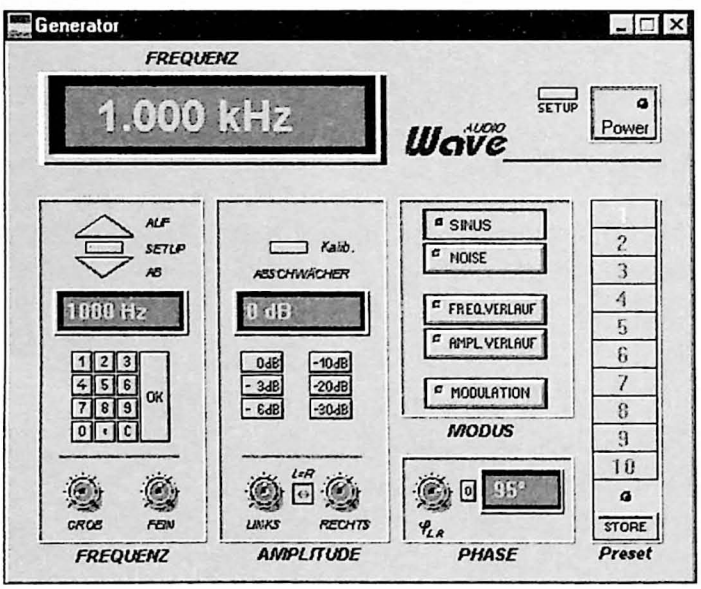


Abb. 3.8: Der einstellbare Sinusgenerator

Frequenzverläufe lassen sich verwenden, um Schaltungen auf Ihren Frequenzgang zu untersuchen. Mit entsprechenden analogen Aufzeichnungsgeräten lassen sich automatisierte Untersuchungen durchführen. Audio-Wave ermöglicht die Einstellung eines linearen oder logarithmischen Verlaufs. Die Auflösung der einzelnen Frequenzschritte und die Geschwindigkeit der Messung lassen sich frei einstellen.

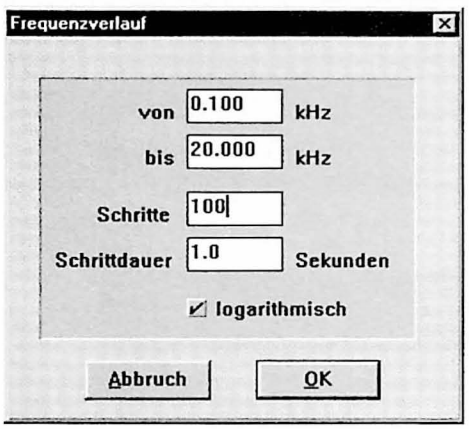


Abb. 3.9: Ein logarithmischer Frequenzverlauf

Das Programm lässt auch eine Amplituden- oder Frequenzmodulation zu. Auch dies lässt sich für automatische Messverfahren verwenden. Man kann z.B. einen einfachen Wobbel-Generator zur Messung von Filterkurven aufbauen. Die Messungen werden im einfachsten Fall mit einem Oszilloskop durchgeführt.

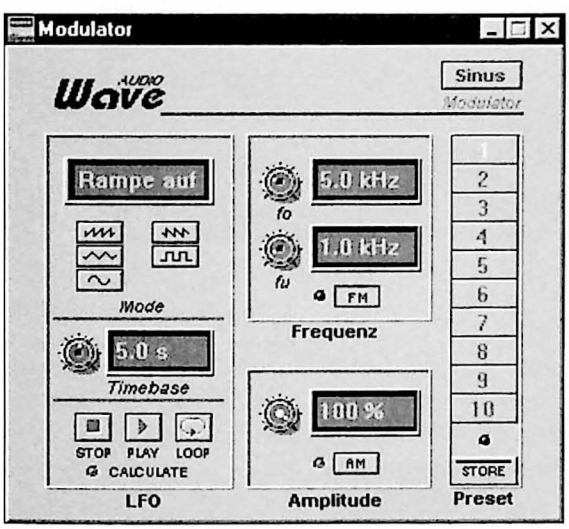


Abb. 3.10: Frequenzmodulation

3.4 Ein Gehörtest

Programme für die Soundkarte arbeiten dank der Windows-Treiber genauso mit einem USB-Soundgerät wie mit einer internen Soundkarte. Viele vorhandene Programme können daher übernommen werden. Die hier vorgestellte Anwendung ist ein Beispiel für die Programmierung der Soundkarte in Delphi. Es stammt aus dem Buch „PC-Schnittstellen unter Windows" (vgl. [6]) und verwendet die PORT.DLL als Bindeglied zwischen Delphi und dem Betriebssystem.

Das menschliche Gehör weist einen ausgeprägten Frequenzgang mit einem Maximum der Empfindlichkeit bei etwa 1000 Hz bis 2000 Hz auf. Mögliche Gehörschäden lassen sich feststellen, indem man die Empfindlichkeit über die Frequenz misst.

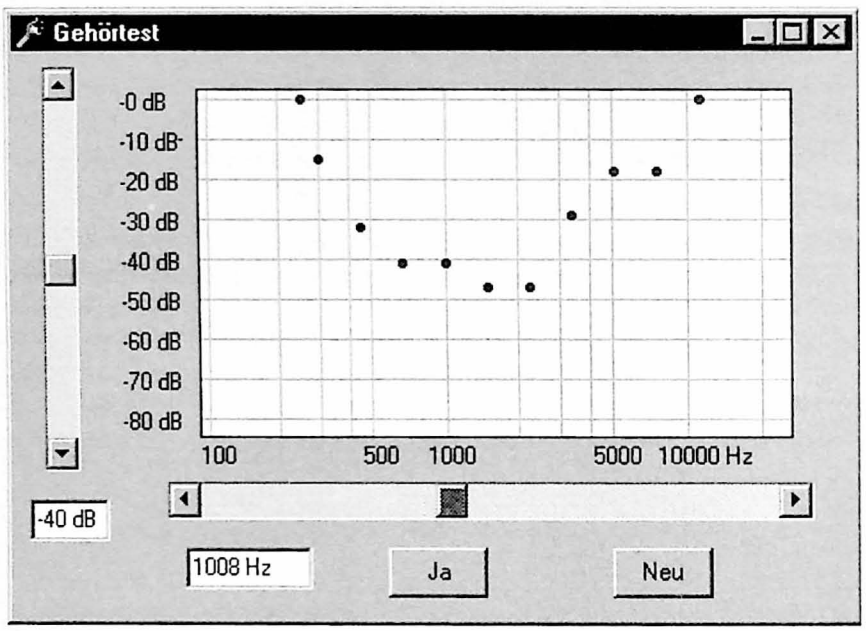


Abb. 3.11: Der Gehörtest

Das Programm wird verwendet, indem man für unterschiedliche Frequenzen jeweils den kleinsten, gerade noch hörbaren Pegel einstellt und diesen dann mit der Ja-Schaltfläche in das Diagramm einträgt. Mit dem Master-Lautstärkeregler des Mischpults kann zuvor der Pegel in einen geeigneten Bereich verschoben werden. Im Bereich der größten Empfindlichkeit soll die Hörschwelle noch oberhalb von –80 dB liegen. Die tatsächlichen Lautstärken können je nach Soundkarte und verwendetem

Kopfhörer sehr unterschiedlich ausfallen, so dass nur eine vergleichende Messung möglich ist.

Das Programm arbeitet mit kurzen Ton-Bursts bei einer Timer-gesteuerten Wiederholrate von zwei Tönen pro Sekunde. Die Töne dürfen aber nicht einfach scharf ein- und ausgeschaltet werden, weil sich dabei ein gewisser Anteil an Signalen mit höherer Frequenz bilden würde. Dies wäre besonders bei der Untersuchung mit sehr tiefen Frequenzen störend, weil die höheren Spektralanteile des Signals in einen Bereich mit besserer Empfindlichkeit des Ohrs fallen würden. Die Sinussignale werden deshalb weich ein- und ausgeblendet.

Das Programm verwendet sowohl bei der Amplitudeneinstellung als auch bei der Frequenzeinstellung einen logarithmischem Maßstab. Die Position beider Schieberegler stimmt dabei mit der Skalierung des Diagramms überein.

```
unit Ohrtest;

interface

uses PortInc,Windows, Messages, SysUtils, Classes,
  Graphics, Controls, Forms, Dialogs, ExtCtrls, StdCtrls;
type
  TForm1 = class(TForm)
    PaintBox1: TPaintBox;
    ScrollBar1: TScrollBar;
    ScrollBar2: TScrollBar;
    Timer1: TTimer;
    Edit1: TEdit;
    Edit2: TEdit;
    Ja: TButton;
    Neu: TButton;
    procedure Timer1Timer(Sender: TObject);
    procedure NeuClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure JaClick(Sender: TObject);
  end;

var
  Form1: TForm1;
  Diagramm: Boolean;

implementation

{$R *.DFM}

procedure DiagrammInit;
var n: Integer;
```

```
begin
  with Form1.Paintbox1.Canvas do begin
  Pen.Color:=ClBlack;
  Brush.Color:=ClWhite;
  Rectangle(40,10,336,185);
  Pen.Color:=ClLtGray;
  for n:= 0 to 8 do begin
      MoveTo(41,15+n*20);
      Lineto (335,15+n*20);
  end;
  Brush.Color:=ClLtGray;
  for n:= 0 to 8 do begin
    TextOut(0,10+n*20,'-'+FloatToStr(n*10)+' dB');
  end;
  for n:= 0 to 2 do begin
      MoveTo(45+120*n,183); Lineto (45+120*n,10);
      MoveTo(45+120*n+36,183); Lineto (45+120*n+36,10);
      MoveTo(45+120*n+57,183); Lineto (45+120*n+57,10);
      MoveTo(45+120*n+72,183); Lineto (45+120*n+72,10);
      MoveTo(45+120*n+83,183); Lineto (45+120*n+83,10);
  end;
  TextOut(40+00,187,'100');
  TextOut(40+83,187,'500');
  TextOut(35+120,187,'1000');
  TextOut(35+203,187,'5000');
  TextOut(30+240,187,'10000 Hz');
  end;
  Diagramm := true;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
CONST RATE=44100;
VAR Size,t,val:Integer;
    Amp, Freq, Gauss: Real;
    p:pchar;
begin
     if not Diagramm then DiagrammInit;
     Size:=RATE div 4;
     GetMem(p,Size*2);
     Amp := 32000 / exp(ScrollBar2.Position/20*ln(10));
     Edit2.Text := '-'+FloatToStr(ScrollBar2.Position)+' dB'
     Freq := round (100* exp(ScrollBar1.Position/12*ln(2)));
     Edit1.Text := FloatToStr(Freq)+ ' Hz';
     For t:= 0 to Size -1 do
     BEGIN
          Gauss := exp(-1*(sqr((t-Size/2)/Size*4)));
          val:=Round(0+Amp*Gauss*sin(2*pi*freq*t/RATE));
          p[t*2]:=chr(LO(val));
          p[t*2+1]:=chr(HI(val));
```

```
    end;
    SoundSetRate(Rate);
    SoundSetBytes(2);
    SoundOut(p,Size);
    FreeMem(p);

end;

procedure TForm1.NeuClick(Sender: TObject);
begin
DiagrammInit;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
Diagramm := false;
end;

procedure TForm1.JaClick(Sender: TObject);
var X, Y: Integer;
begin
  with Form1.Paintbox1.Canvas do begin
    Pen.Color:=ClBlack;
    Brush.Color:=ClRed;
    X:=3*ScrollBar1.Position+45-2;
    Y:=2*ScrollBar2.Position+15-2;
    Ellipse (X,Y,X+5,Y+5);
  end;
end;

end.
```

Listing 3.2: Der Hörtest

Das vollständige Programm liegt in kompilierter Form auf der CD zu diesem Buch bei.

3.5 Ein USB-Joystick-Port

Auch mit einem USB-Gameport lassen sich bereits einfache Versuche durchführen. Wie ein normaler Gameport des PC lässt sich auch die USB-Variante für diverse Aufgaben im Bereich Messen, Steuern und Regeln einsetzen. Das verwendete Gerät der Firma Boeder (vgl. *Abb. 3.12*) enthält einen Mikrocontroller CY7C63000 von Cypress. Dieser Controller wird auch in den folgenden Kapiteln für eigene Geräte eingesetzt.

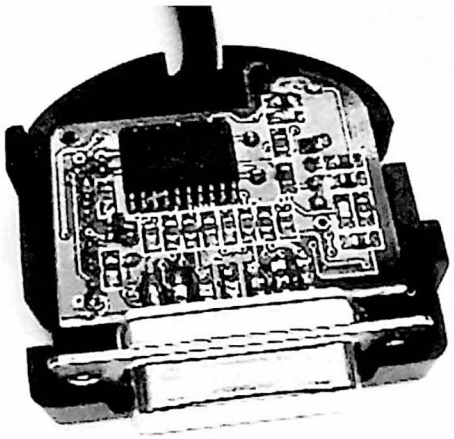


Abb. 3.12: Der geöffnete USB-Gamecontroller

Prinzipiell stellt jeder Gameport vier digitale Eingänge und vier analoge Widerstands-Eingänge zur Verfügung (vgl. *Abb. 3.13*). Zahlreiche Versuche am Gameport wurden bereits in [6] vorgestellt. Man hat mit einem USB-Gerät überdies den Vorteil der bequemen Verbindung, weil man die Verbindungen nicht mehr an der Rückseite des PC vornehmen muss.

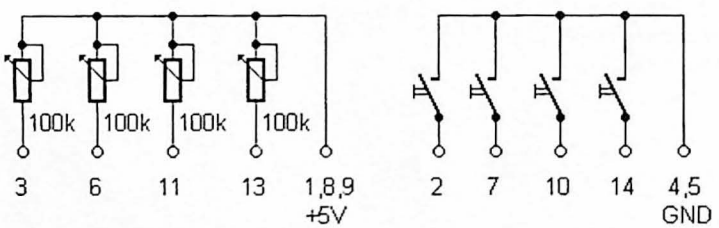


Abb. 3.13: Anschlüsse des Gameports

Ein USB-Game-Controller installiert sich über den Plug-And-Play-Mechanismus von Windows 98 mit den bereits im System vorhandenen Treibern. Dabei wird ein HID-Gerät angemeldet. Zusätzlich stehen aber noch weitere Treiber zur Verfügung. Darunter befindet sich eine DOS-Emulation, die im System eine normale Game-Karte vorspiegelt. Programme können daher wie gewohnt direkt über IO-Adressen auf die Karte zugreifen.

Der Gameport kann bereits im System-Manager auf seine Funktion überprüft werden. Unter der Registerkarte „Gamecontroller" ist ein einfacher Test möglich. Mit einem realen Joystick kann hier die Abfrage der Poti-Widerstände und der Schaltkontakte getestet werden. Es lassen sich aber auch andere Widerstände anschließen.

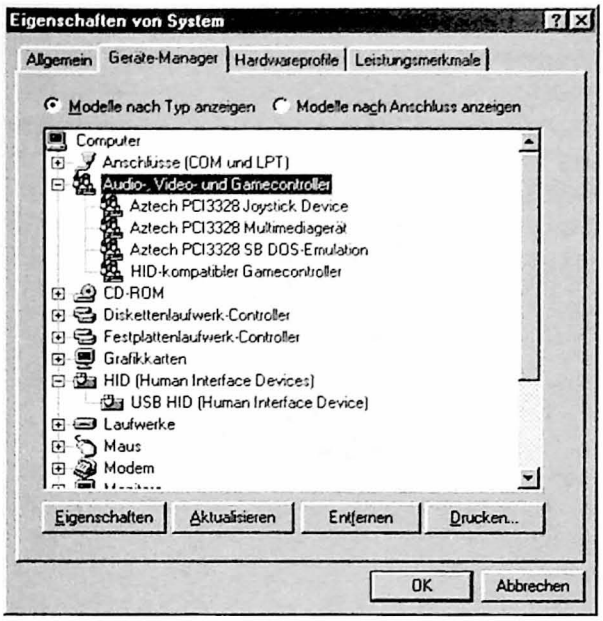


Abb. 3.14: Der USB-Joystick im Gerätemanager

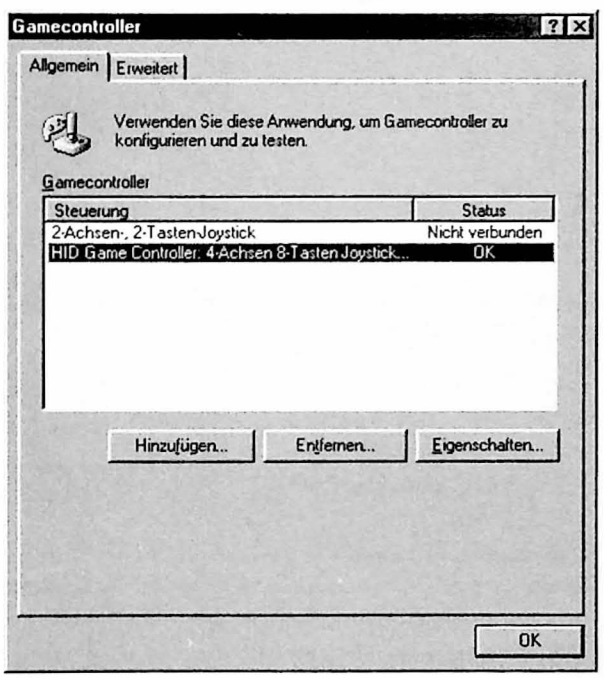


Abb. 3.15: Der Gameport unter System/Gamecontroller

Das Delphi-Programmbeispiel Joy1 (vgl. Listing 3.3) stammt aus [6]. Hier wurde über die Port.DLL direkt auf Portadressen der Karte zugegriffen. Widerstände werden über die Ladezeiten interner Kondensatoren gemessen. Der Treiber für die DOS-Emulation setzt die real gemessenen Werte in ein entsprechendes Verhalten der Joystick-Portadresse 201h um. Dieses Programm demonstriert einen einfachen Zugriff auf vier analoge Eingänge. In der Ausgabe erhält man Ladezeiten in Mikrosekunden (vgl. Abb. 3.16). Das Verfahren kann z.B. zur Abfrage von Widerstands-Sensoren eingesetzt werden. Das ausführbare Programm befindet sich auf der CD.

```
unit Joy1;

interface

uses
  PORTINC,//Enthält die Deklarationen der PORTPAS.DLL
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Timer1: TTimer;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Edit4: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    procedure Timer1Timer(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
function Zaehler (Kanal : Integer) : Word;
var Portwert : Byte;
begin
  Portwert := 1;                        { A0 }
  if Kanal=2 then Portwert := 2;        { A1 }
```

```
  if Kanal=3 then Portwert := 4;        { A2 }
  if Kanal=4 then Portwert := 8;        { A3 }
  RealTime (true);
  TimeInitus;
  OutPort($201,0);                      { Timer-Reset }
  While((InPort ($201) and Portwert) = PortWert)
    and (TimeReadus < 10000)do;
  Zaehler:=TimeReadus;
  RealTime (false);
end;

procedure TForm1.Timer1Timer(Sender: TObject);
var Ausgabe:String;
    Zeit: DWord;
begin
  Edit1.Text := FloatToStr(Zaehler(1)) + ' us';
  Edit2.Text := FloatToStr(Zaehler(2)) + ' us';
  Edit3.Text := FloatToStr(Zaehler(3)) + ' us';
  Edit4.Text := FloatToStr(Zaehler(4)) + ' us';
end;

end.
```

Listing 3.3: Abfrage der Ladezeiten der Joystick-Potis

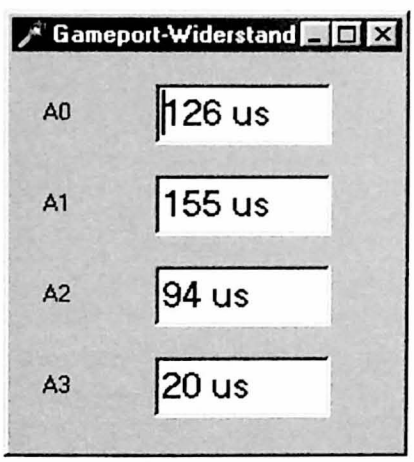


Abb. 3.16: Analogmessung am Gameport

Unter der Adresse 201h lassen sich direkt die vier digitalen Eingänge der Gameports abfragen. Man hat hier bereits ein einfaches USB-Interface mit vier digitalen Eingängen. Das Programm Joy2 (vgl. Listing 3.4) stammt ebenfalls aus [6]. Es kann direkt von der CD ausgeführt werden und funktioniert sowohl mit einem normalen

Gameport wie mit einem USB-Gameport. Allerdings ist die Nutzung der vier analogen Eingänge als Digitaleingänge nur mit einer internen Gamekarte möglich, weil die DOS-Emulation des USB-Gamecontrollers keine statischen Eingangszustände überträgt.

```
unit Joy2;

interface

uses PORTINC,
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs,ExtCtrls, StdCtrls;

type
  TForm1 = class(TForm)
    CheckBox1: TCheckBox;
    CheckBox2: TCheckBox;
    CheckBox3: TCheckBox;
    CheckBox4: TCheckBox;
    CheckBox5: TCheckBox;
    CheckBox6: TCheckBox;
    CheckBox7: TCheckBox;
    CheckBox8: TCheckBox;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    Timer1: TTimer;
    procedure Timer1Timer(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Timer1Timer(Sender: TObject);
var Wert: Byte;
begin
  Wert := InPort ($201);
  CheckBox1.Checked := ((Wert And 1) >0);
  CheckBox2.Checked := ((Wert And 2) >0);
  CheckBox3.Checked := ((Wert And 4) >0);
```

```
  CheckBox4.Checked := ((Wert And 8) >0);
  CheckBox5.Checked := ((Wert And 16) >0);
  CheckBox6.Checked := ((Wert And 32) >0);
  CheckBox7.Checked := ((Wert And 64) >0);
  CheckBox8.Checked := ((Wert And 128) >0);
end;

end.
```

Listing 3.4: Abfrage der digitalen Eingänge eines Gamecontrollers

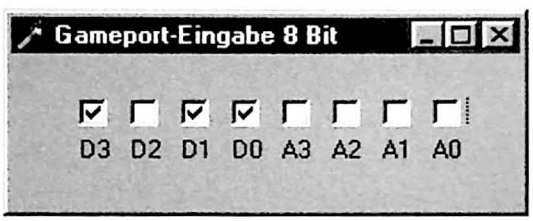


Abb. 3.17: Abfrage der digitalen Eingänge

4 Stromversorgung aus dem USB-Kabel

Eine Besonderheit des USB gegenüber anderen Schnittstellen ist, dass er die Versorgungsspannung für kleinere Geräte gleich mitliefert. Diese Eigenschaft ist interessant, weil damit vielfach die Notwendigkeit eines zusätzlichen Steckernetzteils entfällt. Man kann den USB also zur Stromversorgung z.B. für Mikrocontroller-Systeme verwenden, auch wenn sie selbst über die RS232 mit dem PC kommunizieren. Für erste kleine Versuche sollte man sich einen Anschluss mit einer USB-Buchse vom Typ B herstellen.

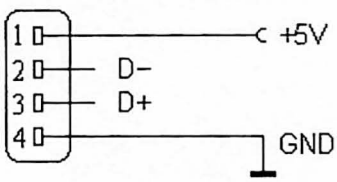


Abb. 4.1: Anschlussbelegung des USB-Kabels

4.1 Belastung und Innenwiderstand

Der Stromversorgungsanschluss kann bis zu 500 mA liefern. Allerdings darf ein Gerät ohne spezielle Anmeldung nur bis zu 100 mA entnehmen. Bei der Enumeration (Anmeldung beim Betriebssystem) kann mehr Strom angefordert werden. Das System entscheidet dann, ob noch genügend Reserven vorhanden sind, und kann im anderen Fall das Gerät zurückweisen. In der Praxis soll ein USB-Port nach den USB-Spezifikationen durch eine Polyswitch-Sicherung automatisch verhindern, dass zu viel Strom entnommen wird. Im Fall einer erfolgreichen Anmeldung mit mehr Strom sollen zusätzliche Halbleiterschalter in Stufen zu je 100 mA weitere Sicherungen parallelschalten. An einem realen PC wurde die Belastbarkeit vorsichtig ausgemessen. Die Ergebnisse der Messung sind in Tab. 4.1 dargestellt. *Abb. 4.2* zeigt die Abnahme der Versorgungsspannung bei zunehmender Belastung.

Tabelle 4.1:Belastungsmessung an der USB-Stromversorgung

I	U
0 mA	5,0 V
100 mA	4,9 V
400 mA	4,6 V

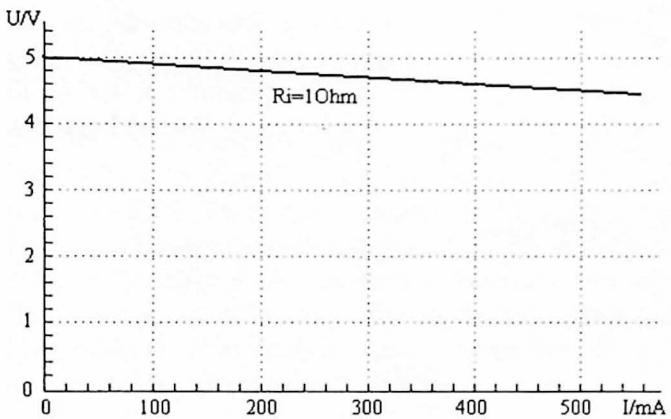


Abb. 4.2: Belastungskennlinie

Die Messergebnisse weisen auf einen Innenwiderstand von 1 Ω hin, der sich auch bei einer Belastung von 400 mA noch nicht erhöht. Demnach ist im untersuchten PC keine Polyswitch-Sicherung eingebaut. Der Innenwiderstand ist im wesentlichen die Summe aller Übergangs- und Kabelwiderstände. Der Kurzschlussstrom dürfte bei 5 A liegen, womit eine Beschädigung des PC nicht mehr auszuschließen wäre. Auf weitere Belastungsexperimente wurde daher verzichtet. Man muss also alles tun, um Überlastungen zu vermeiden.

Dieselben Versuche wurden mit einem USB-Hub wiederholt. Dabei wurde ein Innenwiderstand von 1,6 Ω festgestellt, der sich ebenfalls mit Strömen bis 400 mA nicht veränderte. Also war auch hier kein wirksamer Schutz gegen Kurzschlüsse eingebaut. Diese Aussagen lassen sich nicht verallgemeinern, geben aber Anlass zu erhöhter Vorsicht.

4.2 Absicherung

Will man den USB-Anschluss dauerhaft als kleines Netzteil nutzen, dann sollte eine eigene Sicherung vorgesehen werden. Gut geeignet ist eine Polyswitch-Sicherung. Dieses Bauteil mit PTC-Verhalten zeigt einen sprunghaften Anstieg des Widerstands, wenn ein Überstrom zu einer gewisser Erwärmung führt. Der Vorgang ist reversibel, d.h. die Sicherung wird nach dem Abschalten der Überlastung wieder niederohmig.

Da man meist mit kleinen Strömen auskommt, bietet sich alternativ der Einsatz einer Glühlampe als reversible Sicherung an. Eine Lampe mit 6V/400 mA hat einen Kaltwiderstand von 1,5 Ω, so dass der Innenwiderstand insgesamt nur auf 2,5 Ω steigt. Im Kurzschlussfall wird der Strom jedoch sicher auf unter 400 mA begrenzt.

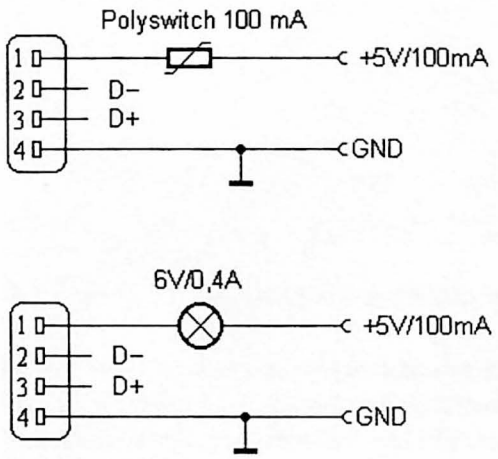


Abb. 4.3: Universelle, abgesicherte USB-Stromversorgung mit Polyswitch-Sicherung oder Glühlampe

4.3 Relaisanschluss

Die am USB bereitgestellte Betriebsspannung reicht auch zum Betrieb eines Relais. Man kann dies ausnutzen, um auf einfache Weise einen Nachteil vieler moderner PCs auszugleichen: Sie besitzen keine geschaltete Ausgangssteckdose für den Monitor mehr. Mit einem Relais lassen sich jedoch sekundäre Verbraucher automatisch mit dem PC einschalten.

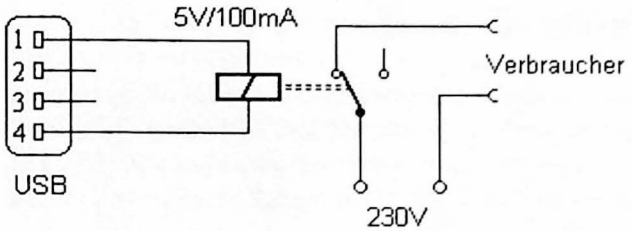


Abb. 4.4: Ein Relais am USB-Anschluss

4.4 Spannungsstabilisierung

In realen USB-Peripheriegeräten arbeitet man meist mit einer Betriebsspannung von 3,3 V. Zum einen hat der Datenbus an den Leitungen D+ und D- Differenzpegel mit 3,3 V. Zum anderen bietet ein 3,3-V-Spannungsregler eine ausreichende Reserve bei Spannnungsabfällen auf längeren Kabeln. Ein guter Regler kann noch bei einem Abfall auf 4 V eine genaue Spannung von 3,3 V abgegeben. *Abb. 4.5* zeigt eine Lösung mit dem Lowpower-Regler LP2950CZ-3.3 im kleinen TO-92-Transistorgehäuse. Aber auch der preiswertere einstellbare Spannungsregler LM317L kann verwendet werden (vgl. *Abb. 4.6*)

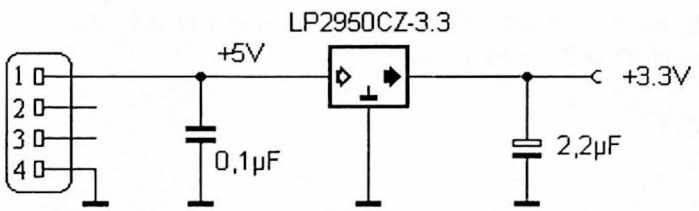


Abb. 4.5: Spannungsregler LP2950CZ3.3

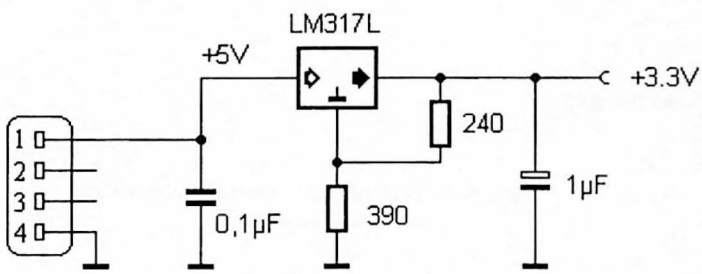


Abb. 4.6: 3,3-V-Regler mit dem LM317L

4.5 Messungen an den Signalleitungen

Ein freier USB-Anschluss zeigt an beiden Signalleitungen D+ und D- die Spannung Null. Man kann einen Innenwiderstand von 15 kΩ messen. Mit einem einfachen Versuch können nun Signale auf dem Bus provoziert werden. Man täuscht dazu den Anschluss eines USB-Geräts vor und beobachtet die Signalleitungen mit einem Oszilloskop.

Jedes USB-Gerät enthält einen Widerstand von 1,5 kΩ, der eine der Leitungen D+ oder D- mit +3,3V verbindet. Die entsprechende Leitung wird also auf ca. 3 V hochgezogen und signalisiert dem PC den Anschluss eines Geräts. Darüber hinaus erfährt er über die verwendete Leitung, ob es ein Fullspeed- (Widerstand an D+) oder ein Lowspeed-Gerät ist (Widerstand an D-). Dementsprechend erfolgt die erste Kontaktaufnahmen des PC mit einer Datenrate von 12 MHz oder von 1,5 MHz.

Man kann diesen Vorgang auch ohne ein wirkliches USB-Gerät provozieren, um die Signale am Oszilloskop zu beobachten. Dazu schaltet man zunächst einen Widerstand von 1,5 kΩ zwischen VCC und D- und beobachtet mit einem Oszilloskop die Signale an einer der Leitungen D+ oder D-. Man erkennt kurze Impulsserien mit einer Übertragungsrate von 1,5 MHz, also mit Pulsabständen von 666 ns. Beim ersten Mal erscheint eine Meldung des PC („Neues Gerät gefunden"). Das System versucht, einen Kontakt zum Peripheriegerät herzustellen und zu erfahren, um welches Gerät es sich handelt. Nach etwa einer Sekunde wird dieser Versuch aufgegeben, weil das nicht vorhandene Gerät nicht antwortet.

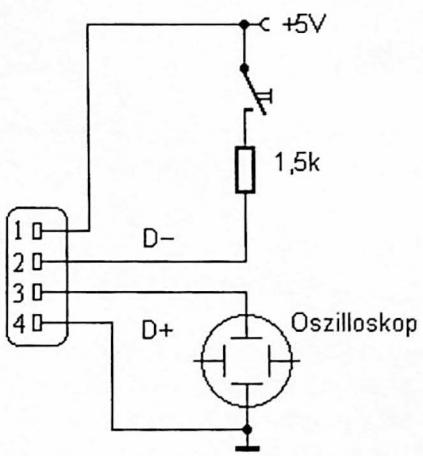


Abb. 4.7: Versuch zur Enumeration eines Lowspeed-Geräts

Führt man denselben Versuch mit einem Widerstand zwischen VCC und D+ durch, wird die erste Kontaktaufnahme mit einer Übertragungsrate von 12 MBit pro Se-

kunde, also mit Impulsabständen von 83,3 ns, eingeleitet. Auch dies ist gut am Oszilloskop zu erkennen.

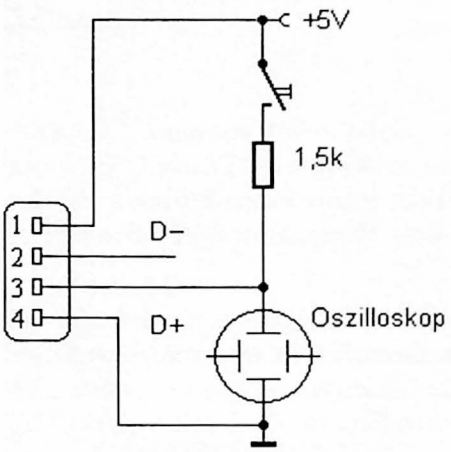


Abb. 4.8: Vortäuschen eines Fullspeed-Geräts

5 Der USB-Controller CY7C63000

Einer der ersten USB-Mikrocontroller für Lowspeed-Devices war der CY7C63000 von Cypress. Der primäre Einsatzbereich dieses Controller sind kleinere USB Geräte wie z.B. die PC-Maus. Entsprechende Beispieldokumentationen befinden sich im PDF-Format auf der CD. Die technische Daten des Mikrocontrollers sollen hier in kurzer Form zusammengefasst werden.

Der Controller enthält einen 8-Bit-RISC Prozessor-Kern mit 128 Byte RAM, 2 oder 4 K OTP-ROM, Timer, Interrupt-Logik und Ports. Zusätzlich ist eine komplette USB-Engine eingebaut. Extern brauchen nur noch die Leitungen D+ und D- an den USB gelegt zu werden. Der Chip benötigt einen externen Keramik-Resonator mit 6 MHz.

Obwohl ein RISC-Prozessor verwendet wird, ist die Programmierung relativ leicht zu erlernen. Der frei erhältliche Assembler CYASM verwendet einen sehr einfachen Code, der für einen an den 8051 gewohnten Programmierer keine besonderen Schwierigkeiten bietet.

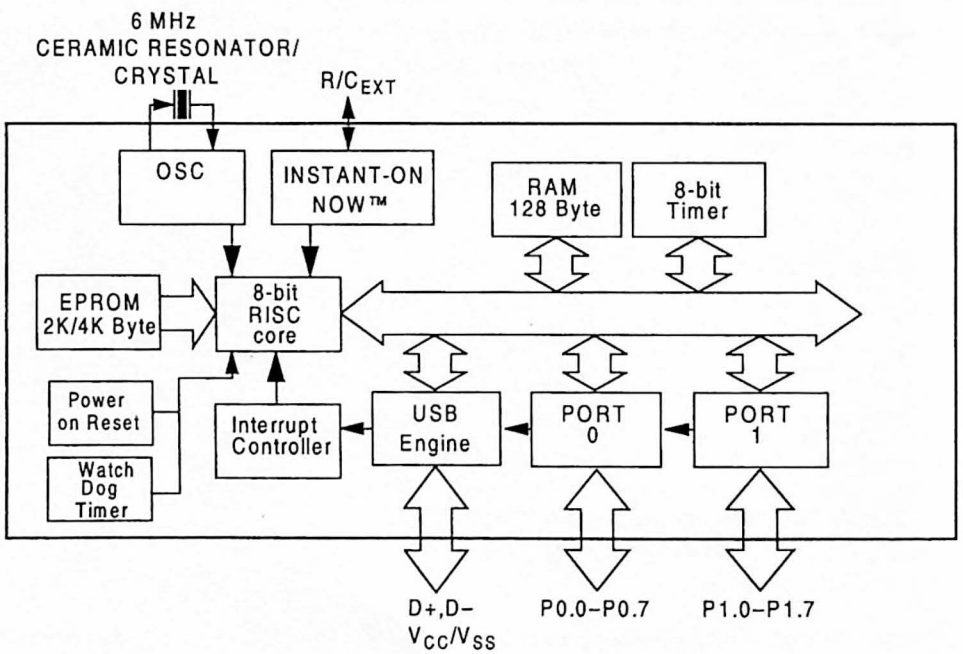


Abb. 5.1: Blockschaltbild des CY7C63000 (Cypress)

Der Controller wird in verschiedenen Varianten mit unterschiedlich vielen heraus geführten Portanschlüssen geliefert. In diesem Buch wird nur der CY7C63001PC im 20-poligen DIP-Gehäuse verwendet. Man hat damit insgesamt 12 Portanschlüsse.

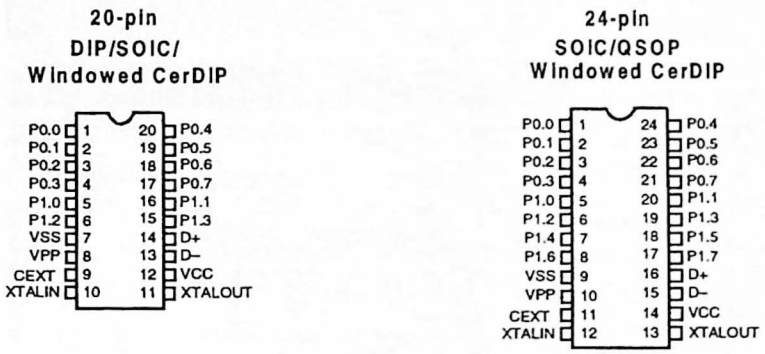


Abb. 5.2: Die Anschlussbelegungen

Die Portanschlüsse des Controllers sind quasi-bidirektional, ähnlich wie beim Mikrocontroller 8051. Eine Besonderheit ist der einstellbare Sink-Strom. Man kann direkt LEDs treiben und deren Strom den Erfordernissen anpassen. Jeder Portanschluss besitzt dazu einen einfachen DA-Wandler mit einer Auflösung von 4 Bit. Jeder Port-0-Pin kann in einem Bereich von 0,3 mA bis 1,5 mA eingestellt werden, für alle Anschlüsse an Port 1 gilt der Einstellbereich 4,8 mA bis 15 mA.

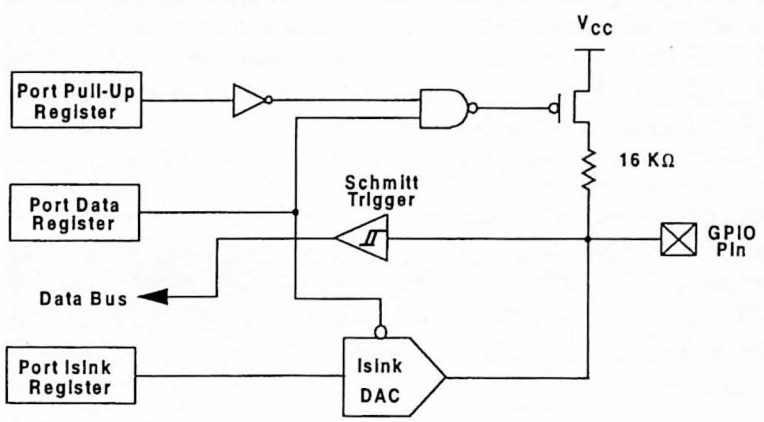


Abb. 5.3: Aufbau eines Portpins mit einstellbarem Strom

5.1 Das USB-Thermometer von Cypress

Das USB-Thermometer der Firma Cypress ist eine Beispiel-Entwicklung zur Vorstellung des Mikrocontrollers CY7C63000. Es wurde im Zusammenhang mit einem Starterkit und im Internet veröffentlicht. Die Anwendung wird hier vorgestellt und analysiert. In den folgenden Kapiteln wird gezeigt, wie man Schritt für Schritt aus dem USB-Thermometer eigene, ganz andere Geräte entwickeln kann. Dieses Vorgehen erlaubt es, einige schwierige Punkte des USB-Entwicklung vorerst zu umgehen. Man fängt also nicht bei Null an, sondern lernt von einem bestehenden Entwurf.

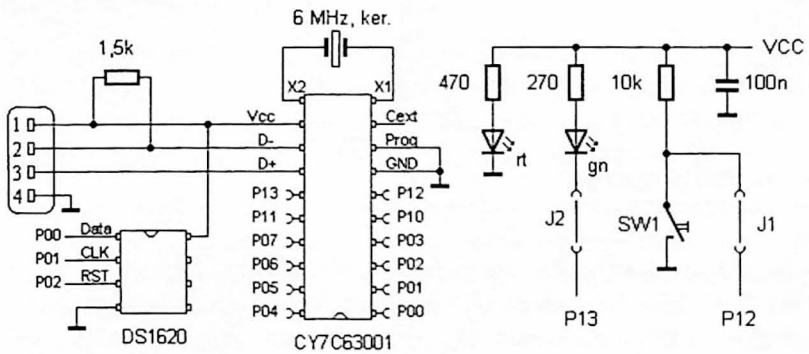


Abb. 5.4: Die Schaltung des USB-Thermometers (Nach Unterlagen von Cypress)

Die Schaltung nach *Abb. 5.4* zeigt den einfachen Anschluss des Mikrocontrollers an den USB. Die Spannungsversorgung beträgt 5 V, so dass kein Spannungsregler erforderlich ist. Der Controller arbeitet mit einem Keramik-Resonator mit 6 MHz. Prinzipiell kann hier auch ein Quarz verwendet werden. Es kann dabei jedoch Probleme mit der längeren Einschwingzeit geben.

Der verwendete Temperatursensor ist ein DS1620 der Firma Dallas. Das IC wird über nur drei Leitungen vom Controller angesteuert. Daher bleiben zahlreiche Portanschlüsse für eigene Anwendungen frei. Zusätzlich gibt es Anschlüsse für alle Ports, ein Tastschalter und zwei LEDs. Eine rote LED zeigt die anliegende Betriebsspannung, eine grüne LED signalisiert die erfolgreiche Enumeration des Geräts und demonstriert zusätzlich die Einstellung des Sink-Stroms eines Portpins. Die Helligkeit der LED kann also verändert werden.

Der Mikrocontroller wird mit der Demo-Firmware USB20a.ASM programmiert. Als Assembler dient Cyasm.Exe, der in der Version 1.77 auf der CD vorliegt. Für das Thermometer existiert ein spezieller Treiber USBtherm.sys und eine zugehörige INF-Datei CypressSemiconductorsCYPRESS.inf. Beim ersten Anschluss des Geräts

erkennt Windows98 eine neue Hardware und installiert den Treiber von einer Diskette.

Die zugehörige Demosoftware Themometer.Exe realisiert ein Thermometer mit grafischer Anzeige des Temperaturverlaufs. Der Anwender kann über den Taster auf dem Board zwischen der Anzeige in Grad Celsius und Fahrenheit umschalten.

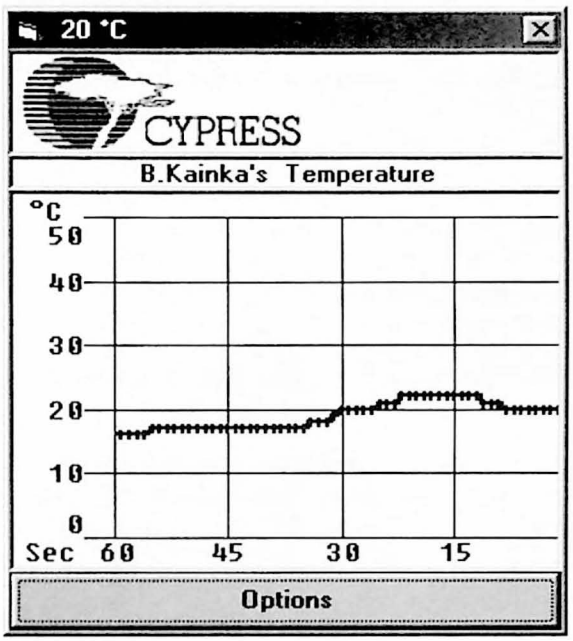


Abb. 5.5: Die Temperaturanzeige

Über die Schaltfläche „Options" erreicht man ein zusätzliches Menü mit weiteren Einstellungen (vgl. *Abb. 5.6*). Hier kann die Helligkeit der grünen LED eingestellt werden. Damit demonstriert Cypress die Möglichkeiten des programmierbaren Portstroms.

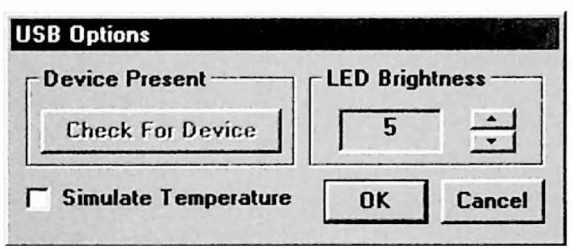


Abb. 5.6: Einstellung des LED-Stroms

5.2 Programmierung in Visual Basic

Für den Hobbyanwender stellt sich die Frage, ob mit dem vorhandenen Chip und dem Treiber von Cypress auch andere Dinge zu machen sind. Notwendig ist dazu ein Zugang über eigene Anwenderprogramme. Cypress dokumentiert ein Grundgerüst für die Ansteuerung des Treibers in Visual Basic. Damit gelingt der erste Kontakt über ein eigenes Programm. In der Datei USB1.BAS wurden die Deklarationen der Windows-Funktionen CreateFile, CloseHandle und DeviceIoControl in Visual Basic umgesetzt. Damit lassen sich beliebige Treiber aus VB heraus ansteuern.

```
Type SECURITY_ATTRIBUTES
nLength As Long
lpSecurityDescriptor As Long
bInheritHandle As Long
End Type

Declare Function CreateFile Lib "kernel32" Alias "CreateFileA"
 (ByVal lpFileName As String, ByVal dwDesiredAccess As Long,
 ByVal dwShareMode As Long, lpSecurityAttributes As
 SECURITY_ATTRIBUTES, ByVal dwCreationDisposition As Long, ByVal
 dwFlagsAndAttributes As Long, ByVal hTemplateFile As Long)
 As Long
Declare Function DeviceIoControl Lib "kernel32" (ByVal hDevice
 As Long, ByVal dwIoControlCode As Long, lpInBuffer As Any,
 ByVal nInBufferSize As Long, lpOutBuffer As Any, ByVal
 nOutBufferSize As Long, lpBytesReturned As Long, lpOverlapped
 As OVERLAPPED) As Long
Declare Function CloseHandle Lib "kernel32" (ByVal hObject
 As Long) As Long
Public Security As SECURITY_ATTRIBUTES
Public gOverlapped As OVERLAPPED
Public hgDrvrHnd As Long
Public Const GENERIC_READ = &H80000000
Public Const GENERIC_WRITE = &H40000000
Public Const FILE_SHARE_WRITE = &H2
Public Const FILE_SHARE_READ = &H1
Public Const OPEN_EXISTING = &H3
```

Listing 5.1: Das Modul USB1.BAS mit Deklarationen

```
Dim sFileName As String
Dim htemp As Long
Dim lIn As Long, lInSize As Long, lOut As Long,
Dim lOutSize As Long, lSize As Long
Dim lTemp As Long
```

```
Private Sub Form_Load()
Timer1.Interval = 1000
End Sub
Private Sub Timer1_Timer()
sFileName = "\\.\Thermometer_0"
hgDrvrHnd = CreateFile(sFileName, GENERIC_WRITE Or GENERIC_READ,
   FILE_SHARE_WRITE Or FILE_SHARE_READ, Security, OPEN_EXISTING,
   0, 0)
' Get Handle to Driver
lIn = 11
lInSize = 2
lOutSize = 3
lTemp = DeviceIoControl(hgDrvrHnd, 4&, lIn, lInSize, lOut,
   lOutSize, lSize, gOverlapped)
' GetTemperature Command
htemp = CloseHandle(hgDrvrHnd)
' Close handle to driver
Text1.Text = Str$(((lOut \ 256) And 255) / 2) + "°C"
End Sub
```

Listing 5.2: Auslesen der Temperatur in TEMP1.FRM

Das Programm liest regelmäßig über eine Timer-Prozedur die Temperatur des Sensors auf dem Starterkit. Dazu wird jedesmal der Treiber „\\.\Thermometer_0" geöffnet, aufgerufen und wieder geschlossen. Beim Aufruf mit DeviceIoControl wird die Treiber-Funktionsnummer (IoControlCode) 4 übergeben.

Dem Treiber werden zwei Variablen lIn und lOut übergeben. Beide sind als Long definiert, haben also eine Länge von vier Bytes. Zur Abfrage der Thermometerfunktion muss eine Funktionsnummer 11 in lIn übergeben werden. Der Treiber liefert dann das Ergebnis in lOut zurück. Die Temperatur befindet sich im Byte 2 und wird als Vielfaches von 0,5 Grad Celsius angegeben. Die zusätzlich gelieferten Informationen zum Vorzeichen der Temperatur und zum Zustand des Tastschalters auf dem Board werden hier nicht ausgewertet.

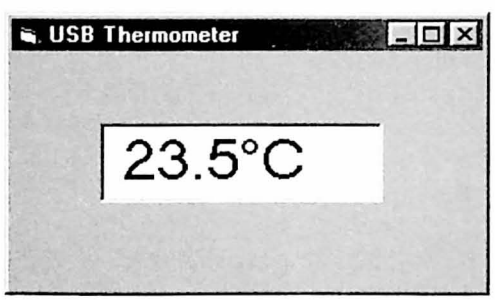


Abb. 5.7: Ausgabe der Temperatur

Die verfügbaren Treiberfunktionen wurden von Cypress dokumentiert. Während die Treiber-Funktionsnummer immer 4 ist, wird die eigentliche Funktion mit einem Steuerbyte in lIn ausgewählt. Der Treiber liefert je nach Funktion zwei bis vier Bytes zurück. Das niederwertigste Byte enthält immer eine Statusinformation zum Erfolg des Zugriffs. Neben dem Auslesen des Thermometers gibt es spezielle Funktionen zum Einstellen der LED-Helligkeit und für Zugriffe auf Speicher und Ports des Controllers.

0Bh: Read Thermometer
lIn: 0Bh (Länge: 1 Byte)
iOut: Button, Vorzeichen, Temperatur, Status (Länge: 4 Bytes)

0Eh: Set LED Brightness
lIn: Helligkeit (0...15), 0Eh (Länge: 2 Bytes)
iOut: Status (Länge: 1 Byte)

14h: Read Port
lIn: Port (0,1), 14h (Länge: 2 Bytes)
iOut: Wert, Status (Länge: 2 Bytes)

15h: Write Port
lIn: Wert (0...255), Port (0,1), 15h (Länge: 3 Bytes)
iOut: Status (Länge: 1 Byte)

16h: Read RAM
lIn: Adresse (0...255), 16h (Länge: 2 Bytes)
iOut: Wert, Status (Länge: 2 Bytes)

17h: Write RAM
lIn: Wert (0...255), Adresse (0...255), 17h (Länge: 3 Bytes)
iOut: Status (Länge: 1 Byte)

18h: Read ROM
lIn: Index, Adresse (0...255), 18h (Länge: 3 Bytes)
iOut: Wert, Status (Länge: 2 Bytes)

5.3 Einstellung des LED-Stroms

Die Verwendung spezieller Treiberfunktionen soll hier an einem Beispiel erläutert werden. Ein kleines Testprogramm soll die Helligkeit der LED über einen Schiebe-

regler verstellen. Hier wird das Kommando 14h verwendet. Der Wertebereich des Schiebers soll auf 0 bis 15 eingestellt werden.

```
Private Sub VScroll1_Change()
sFileName = "\\.\Thermometer_0"
lIn = 14 + 256 * VScroll1.Value 'LED Brightness
lInSize = 2
lOutSize = 1
hgDrvrHnd = CreateFile(sFileName, GENERIC_WRITE Or GENERIC_READ,
FILE_SHARE_WRITE Or FILE_SHARE_READ, Security, OPEN_EXISTING, 0, 0)
lTemp = DeviceIoControl(hgDrvrHnd, 4&, lIn, lInSize, lOut, lOutSize,
lSize, gOverlapped)
htemp = CloseHandle(hgDrvrHnd)
End Sub
```

Listing 5.3: Einstellung des Ausgangsstroms in LED.FRM

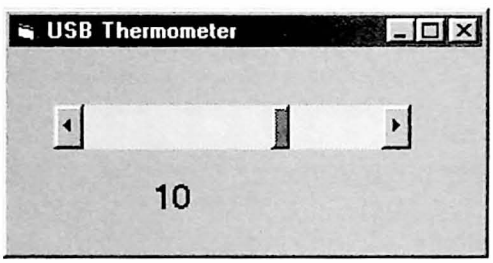


Abb. 5.8: Das Programm LED.FRM

Der Mikrocontroller enthält für jeden Portanschluss einen einfachen 4-Bit-DAC aus vier gewichteten Stromquellen. Es stellt sich die Frage, ob man damit nicht mehr anfangen kann, als nur die Helligkeit von LEDs zu steuern. Im Prinzip reicht ein Widerstand, um den gesteuerten Strom in eine Ausgangsspannung umzusetzen. Mit einem Widerstand von 220 Ω ergibt sich eine gute Aussteuerung bei einem Sink-Strom bis 15 mA.

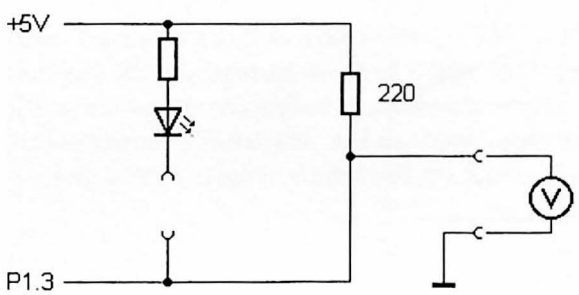


Abb. 5.9: Steuern einer Ausgangsspannung über P1.3

Zur Überprüfung der Linearität wurden die Ausgangsspannungen für alle 16 möglichen Stromeinstellungen gemessen. Tabelle 5.1 zeigt die Ergebnisse. Die Messwerte wurden auch grafisch ausgewertet. *Abb 5.10* zeigt eine gute Linearität.

Tabelle 5.1:Die gemessenen Spannungen an P1.3

0	4,29 V
1	4,09 V
2	3,90 V
3	3,70 V
4	3,51 V
5	3,32 V
6	3,15 V
7	2,96 V
8	2,79 V
9	2,61 V
10	2,43 V
11	2,24 V
12	2,06 V
13	1,89 V
14	1,71 V
15	1,54V

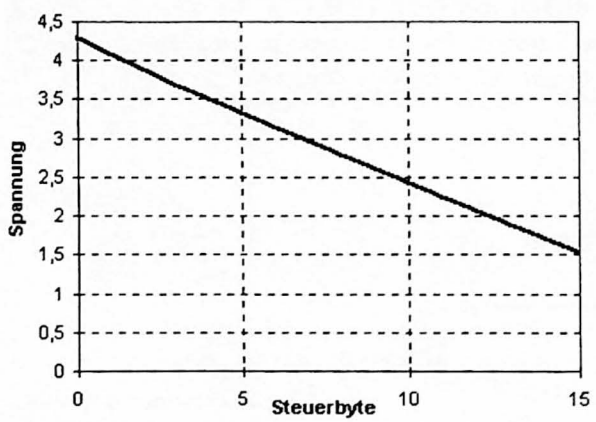


Abb. 5.10: Ausgangsspannung, aufgetragen gegen die Steuerbytes

Der Strom-Ausgang kann für ein einfaches steuerbares Netzteil verwendet werden. *Abb. 5.11* zeigt eine Schaltung mit einem Power-OPV L272. Nullpunkt und Steigung können getrennt eingestellt werden. Je nach Bedarf läßt sich also auch eine kleinere Variation, z.B. zwischen 3,5 V und 5 V erreichen.

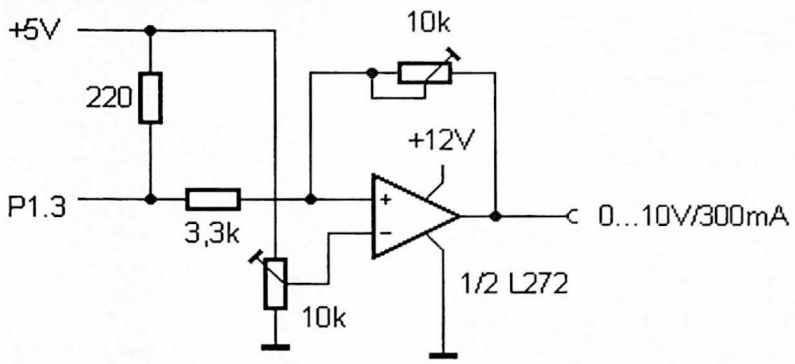


Abb. 5.11: Eine gesteuerte Spannungsquelle

5.4 Weitere Treiberfunktionen

Alle Aufrufe und Funktionen des Treibers lassen sich mit einem kleinen Hilfsprogramm Fiddler.Exe untersuchen, das von Craig Peacock geschrieben wurde. Der „Device Driver Fiddler" öffnet den Treiber und führt beliebige Control-Zugriffe aus. Die einzelnen Funktionen des Treibers können auf einfache Weise getestet werden.

Abb. 5.12 zeigt die Ausführung des Set-Brightness-Kommandos. Übergeben wird die Helligkeit (07h) und das Funktions-Steuerbyte 0Eh. Der Treiber liefert zwei Bytes zurück. Im LSB steht die Statusmeldung 0 für die erfolgreiche Bearbeitung. Das zweite Byte liefert die Helligkeit zurück.

Beim Testen fällt auf, dass die Funktion 15h (Write Port) nicht wie gewünscht funktioniert. Alle Ausgaben an Port 0 werden vom Controller nach wenigen Millisekunden wieder verändert. Ausgaben an Port 1 werden nicht ausgeführt. Dagegen funktionieren Lesezugriffe auf die Ports einwandfrei. Die Ursache der nicht funktionierenden Ausgaben findet sich im Assemblerprogramm des Controllers.

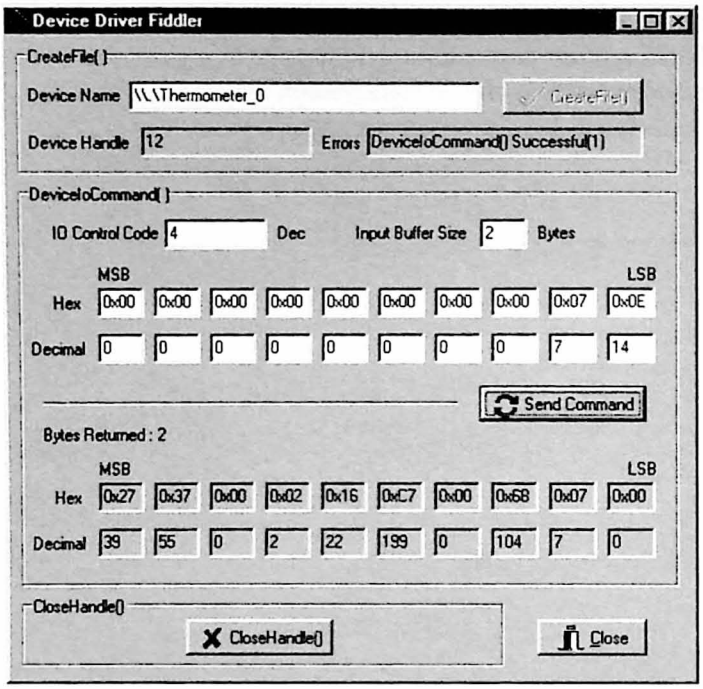


Abb. 5.12: Treiberzugriff mit Fiddler.Exe

5.5 Analyse der USB-Datenübertragung

Das Assemblerprogramm USB_20a.asm leistet gute Dienste bei der Analyse der Kommunikation zwischen Treiber und Hardware. Der vollständige Quelltext befindet sich auf der CD. Das genaue Studium des Programms lohnt sich für jeden, der selbst USB-Firmware schreiben will. Die folgenden Ausführungen sollen den Überblick erleichtern.

Das Controller-Programm richtet zwei Endpoints ein, EP 0 für Enumeration und Vendor-spezifische Control-Requests, EP 1 für Interrupt-Requests. Allerdings wird EP 1 nicht wirklich verwendet. Die Enumeration mit EP 1 ist also reine Formsache. Nach der USB-Philosophie ist EP 0 nur für die Installation und für die Initialisierung bestimmter Grundeinstellungen zuständig, nicht aber für den laufenden Datenaustausch. Ein USB-Gerät muss also mindestens einen Endpoint 0 und einen Endpoint 1 besitzen. Dass dann im laufenden Betrieb die eigentlichen Nutzdaten trotzdem nur über EP 0 ausgetauscht werden, ist für das Betriebssystem kein Problem.

Alle USB-Aktionen werden von entsprechenden Interrupt-Routinen durchgeführt. Die Vektor-Tabelle nach Listing 5.4 zeigt, welche der möglichen Interrupts verwendet werden. Hier kann man sehen, wie eine Interrupt-Serviceroutine des Prozessors aufgebaut sein muss. Alle nicht verwendeten Interrupts werden zur Sicherheit durch das Unterprogramm SysUnUsed abgearbeitet.

```
; Vector Table
org 00h
    jmp main                ; Reset of some type
    jmp SysUnUsed           ; 128us timer (not used)
    jmp SysTimer1024usEvent ; 1024us timer
    jmp USBEndPoint0Event   ; EP0
    jmp SysUnUsed           ; EP1 (not used)
    jmp SysUnUsed           ; Reserved
    jmp SysGPIOEvent        ; Button
    jmp SysUnUsed           ; CExt (not used)

;***********************************************
; Unused event
; Do nothing, restore machine to prior state
;***********************************************
SysUnUsed:
    push  a
    mov   a,[gbSysInterruptMask]
    ipret SysInterrupt
```

Listing 5.4: Interruptroutinen

Immer wenn der PC einen Control-Request absendet, erfolgt ein Endpoint-0-Interrupt. Dies geschieht zuerst bei der Enumeration. Sobald Windows ein neues Gerät erkannt hat, werden hier seine Eigenschaften abgefragt. Der Controller sendet auf entsprechende Anfragen seine Deskriptoren und erhält bei Erfolg eine Device-Nummer. Damit ist die Enumeration abgeschlossen. In der Interrupt-Routine wird der Wert 1 in die globale Systemvariable gbSysEnumerated (Adresse 29h im RAM) geschrieben. Allgemein beantwortet die Interruptroutine des Endpoint 0 auch alle folgenden Requests.

Alle Anfragen des PCs lösen im Controller Ereignisse aus, die mit entsprechenden Aktionen beantwortet werden. Dabei ist grundsätzlich der folgende Ablauf gegeben: Vom PC wird ein Datenpaket gesendet, das von der SIE (Serial Interface Engine) in den FIFO-Speicher des entsprechenden Endpoints geladen wird.

Jeder Endpoint-0-Request führt zu einem Endpoint-0-Interrupt. Der Prozessor springt die Interrupt-Serviceroutine USBEndPoint0Event an und beantwortet die Anfrage. Dazu wird zunächst das Byte an der Position 0 im FIFO gelesen. Es gibt an,

um welche Art von Anfrage es sich handelt. Die meisten Anfragen werden während des Enumerations-Prozesses benötigt. Der Prozessor wird z.B. aufgefordert, seine Deskriptoren zu senden, seine Adresse im USB-Baum entgegenzunehmen usw.. Die angeforderten Daten werden grundsätzlich wieder in denselben FIFO-Speicher zurückgeschrieben. Das Programm gibt dann ein Kommando zum Absenden und wartet, bis die Aussendung bestätigt wurde. Je nach Menge der zu sendenden Daten kann es vorkommen, dass ein Deskriptor in mehrere Datenpakete aufgeteilt werden muss.

Die erforderlichen Deskriptoren sind im ROM abgelegt und werden von dort in das FIFO kopiert. Der USBDeviceDescriptor hat z.B. eine Länge von 18 Bytes. Die Länge selbst ist an der Position 0 gespeichert. Der Deskriptor enthält u.a. die Hersteller ID, die Produkt ID und die Versionsnummer, die jeweils zwei Bytes belegen. Er muss entsprechend auf drei Pakete aufgeteilt werden.

Besonders wichtig für den Betrieb als Messgerät sind hier die Vendor-Requests, also die von Hersteller der Hardware festgelegten internen Anfragen. Über diese läuft hier der eigentliche Datenverkehr ab. Der PC kann z.B. Daten im RAM und an den Ports lesen und verändern. Dies soll an einem Beispiel verdeutlicht werden. Die Endpoint-0-Interruptroutine hat einen Vendor-Request erkannt und verzweigt zum Abschnitt ESBEventEP0VendorRqst. Dort wird das Byte an der Position 1 im FIFO als Funktionsnummer ausgewertet. Findet sich hier eine 2, dann soll das RAM gelesen werden. Es wird nun an der FIFO-Position 2 die gewünschte RAM-Adresse gelesen und damit die Anfrage ausgeführt. Das Ergebnis ist ein gelesenes Byte, das in diesem Fall an die Position 1 in das Endpoint-0-FIFO zurückgeschrieben wird. Dann wird der gesamte FIFO-Inhalt zurückgesandt. Der Treiber entnimmt aus dem Datenpaket das gewünschte Byte und gibt es an ein Anwenderprogramm zurück.

Die Kommunikation zwischen Anwenderprogramm, Treiber, USB und Prozessor beruht beim USB-Thermometer zu einem großen Teil auf dem direkten Zugriff auf RAM-Adressen des Prozessors. Zum Verständnis der Firmware ist deshalb der Aufbau des Prozessor-RAMs und die Belegung mit anwendungsspezifischen Systemvariablen wichtig. Listing 5.5 zeigt den Aufbau.

```
;***********************************************
; Data Segment (RAM)
;***********************************************

; Program Stack
gbSysProgramStack       :equ 00h    ; [00h-1Fh] Stack 0x20h
gbSysDataStack          :equ 50h    ; [50h-6Fh] Stack 0x20h
gbSysFIFO               :equ 70h    ; [70h-7Fh] EP0 and EP1 FIFO's

; Global Interrupt
gbSysInterruptMask      :equ 20h    ; Holds the current interrupt mask
```

```
; System tickers
gbSysTick1024us         :equ 22h    ; # of 1mSec ticks
gbSysTick1024usRoll     :equ 24h    ; # of 256mSec ticks

; USB management data
gbUSBValidRqsts         :equ 25h    ; Count of USB recognized requests
                                    ;   Used during debug
gbUSBSendSequence       :equ 26h    ; Buffer send data 0/1 line
gbUSBSendBytes          :equ 27h    ; Buffer bytes left to send
gbUSBSendBuffer         :equ 28h    ; Offset into current buffer
gbSuspendCount          :equ 30h    ; # of msec bus has been IDLE

; General
gbSysEnumerated         :equ 29h    ; Device is enumerated

; LED management
gbLEDBrightnessUpdate   :equ 2Bh    ; Semaphore to reset brightness
gbLEDBrightness         :equ 2Ch    ; Current brightness
LED_ON                  :equ 08h    ; P13 is used to indicate Enumerati-
on

; Button management
gbButtonDebounce        :equ 2Dh    ; Debounce count down value
gbButtonPushed          :equ 7Ah    ; USBEndP1FIFO +2 (toggles if button
was clicked)
Button_Pin              :equ 04h    ; Pin the switch is on, P12
```

Listing 5.5: Belegung der Systemvariablen in USB20a.ASM

Die Software ist in mehrere Funktionsblöcke eingeteilt. Alle USB-Aktivitäten werden von der entsprechenden Interrupt-Routine abgearbeitet. Die Taste löst einen eigenen Interrupt aus. Die Verwaltung des LED-Zustands und der LED-Helligkeit wird im Hauptprogramm durchgeführt. Von hier aus wird auch ein Unterprogramm zum Auslesen der Temperatur aufgerufen.

Das eigentliche Hauptprogramm wartet zunächst nur, bis es die erfolgreiche Enumeration an der entsprechenden Variablen erkennt. Dann schaltet es die grüne Status-LED an und ruft im folgenden alle 10 Millisekunden immer wieder die Thermometer-Routine auf. Diese ist als ein in sich geschlossenes Programm-Modul (DS1620a.asm) geschrieben und liefert zwei Bytes in entsprechenden Systemvariablen zurück: gbThermRead (33h) und gbThermRead2 (34h). Die beiden Ergebnisbytes werden zusätzlich in zwei Adressen des Endpoint-1-FIFOs geschrieben. Da EP 1 nicht verwendet wird, steht der FIFO-Speicher als allgemeines RAM zur Verfügung.

```
gbThermProtocol             :equ 30h     ;
gbThermPortValue            :equ 31h     ;
gbThermPortMirror           :equ 32h     ;
gbThermTempRead             :equ 33h     ;
gbThermTempRead2            :equ 34h     ;
gbThermTempLast             :equ 78h     ;USBEndP1FIFO
gbThermTempLast2            :equ 79h     ;USBEndP1FIFO +1
```

Listing 5.6: Variablen im Modul DS1620a.ASM

Völlig unabhängig vom Hauptprogramm wird die Taste des Systems behandelt. Bei jedem Tastendruck wird Port1.2 herunter gezogen und löst einen GPIO-Interrupt aus. Die Timer-Interruptroutine sorgt für eine Entprellung des Schalters. Im Endergebnis wird die Systemvariable gbButtonPusched (7Ah) umgeschaltet. Der Wert dieser Variablen wechselt mit jedem Tastendruck zwischen 0 und 1.

Die wichtigsten Zugriffe betreffen den USB-Vendor-Request. Sie werden vom Treiber abgesandt, sobald ein Anwenderprogramm den ControlCode 4 übergibt. Im Quelltext der Firmware findet sich hierzu der Abschnitt USBEvntEP0VendorRqst. Der Controller erhält hier in Endpoint-0-FIFO an der Position 1 eine Funktionsnummer:

```
Ping            0x00
Read ROM        0x01
Read RAM        0x02
Write RAM       0x03
Read Port       0x04
Write Port      0x05
```

Jede Funktion erwartet spezifische Informationen an bestimmten Positionen des FIFO-Puffers. Leseaktionen liefern entsprechende Bytes zurück. Die Ping-Funktion dient nur zum Test der grundlegenden Funktion des Controllers.

Auch das Auslesen des Thermometers durch den Treiber erfolgt über das Lesen bestimmter RAM-Adressen, d.h. der Treiber schaut direkt ins RAM und holt sich, was er braucht. Die Treiberfunktion ReadTemperature kann daher auch durch drei eigene ReadRAM-Anfragen ersetzt werden.

Auch eine Zusatzfunktion der Anwendung, nämlich das Steuern der LED-Helligkeit, wird über das RAM gesteuert. Dazu gibt es zwei Systemvariablen: gBBrightnessUpdate (2Bh) und gBBrightness (2Ch). Zum Ändern der Helligkeit schreibt man einen neuen Wert (0....15) in gbBrightness und danach eine 1 in gBBrightnessUpdate als Semaphore, also als Wink an den Controller. Das Programm erkennt den Wunsch nach einem Update der Helligkeit in der MainLoop-Schleife des Hauptprogramms und überträgt den gewünschten Wert in das Steuerregister SysPort1SinkPin3. Außerdem wird gBBrightnessUpdate wieder auf Null zurückgesetzt.

5.6 Portausgaben

Während das Thermometer, die Taste und die Helligkeitssteuerung für die Status-LED wie gewünscht funktionieren, werden Portausgaben über den Treiber nicht ordnungsgemäß ausgeführt. Hier liegt jedoch der eigentliche Reiz jedes Mikrocontrollers, denn mit Portausgaben erschließen sich zahlreiche Möglichkeiten.

Es wurde ein Weg gesucht, bei unverändertem Treiber und nur durch kleine Änderungen an der Firmware des Controllers Portausgaben zu ermöglichen. Die Lösung fand sich in der Verwendung des WriteRAM-Kommandos. Zwei zusätzliche RAM-Variablen wurden definiert:

```
;Portmanagement
gbPort0                 :equ 2Eh
gbPort1                 :equ 2Fh
```

Der Anwender kann nun direkt in diese Variablen schreiben. Die eigentliche Ausgabe an die physikalischen Ports wurde mit in der Interruptroutine SysTimer1024us Event() untergebracht. Die Portleitungen P0.0 bis P0.2 werden nicht verändert, um die Thermometerfunktion nicht zu stören. Über diese Leitungen greift das Programm auf den Sensor DS1620 zu.

```
STPortsRefresh:    ;neu
    mov  a,[gbPort0]
    and a,0F8h
    mov [gbPort0],a
    iord SysPort0
    and  a,07h
    or     a,[gbPort0]
    iowr SysPort0
    mov  a,[gbPort1]
    iowr SysPort1
```

Listing 5.7: Die veränderten Portausgaben

Zusätzlich war es erforderlich, den neuen Systemvariablen einen Startwert zu geben.

```
    mov  a,FFh
    iowr SysPort0    ; Port 0 Data reg
    iowr SysPort1    ; Port 1 Data reg
    mov [gbPort0],a      ;neu
    mov [gbPort1],a
```

Listing 5.8: Initialisierung der Ports

Die ursprüngliche Anwendung schaltete die LED ein, sobald die Enumeration erfolgreich beendet war. Die Abfrage des Zustands erfolgte jedoch in der Hauptschleife des Programms. Eine Änderung der Zustände an Port 1 wurde damit regelmäßig wieder zurückgenommen. Die Schleife wurde daher geteilt in eine MainLoop und eine Loop2. Mainloop wird nur so lange durchlaufen, bis die Enumeration erkannt wurde. Danach wird nur noch die Schleife Loop2 ausgeführt, in der nicht mehr auf den Port 1 zugegriffen wird. Die Zustands-LED wird also beim Anschluss der Geräts eingeschaltet, um den Erfolg zu signalisieren. Danach aber kann der Anwender den Port für sich verwenden. Das abgeänderten Programm USB_20e.asm befindet sich auf der CD.

Das Programm USBports.FRM dient dem Zugriff auf alle Portfunktionen des Controllers. Zusätzlich sollten alle ursprünglichen Funktionen des Thermometers erhalten bleiben. Das Programm arbeitet weiterhin mit dem USB-Treiber USBTherm.sys.

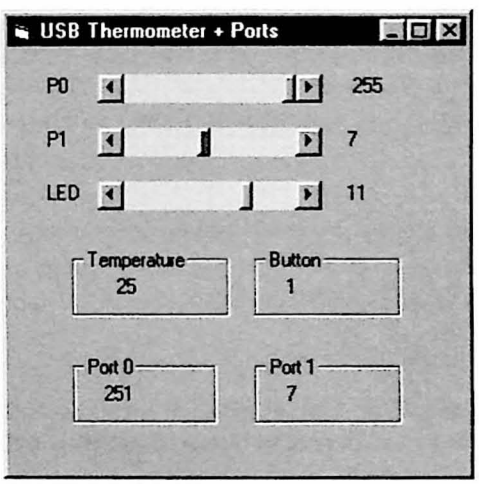


Abb. 5.13: Portzugriffe mit USBports.FRM

```
Dim sFileName As String
Dim htemp As Long
Dim lIn As Long, lInSize As Long, lOut As Long,
Dim lOutSize As Long, lSize As Long
Dim lTemp As Long
Sub USB_IO()
  sFileName = "\\.\Thermometer_0"
  hgDrvrHnd = CreateFile(sFileName, GENERIC_WRITE
     Or GENERIC_READ, FILE_SHARE_WRITE Or FILE_SHARE_READ,
     Security, OPEN_EXISTING, 0, 0)
  lTemp = DeviceIoControl(hgDrvrHnd, 4&, lIn, lInSize, lOut,
     lOutSize, lSize, gOverlapped)
  htemp = CloseHandle(hgDrvrHnd)
End Sub
```

```
Sub Brightness(Level)
  lIn = Level * 256 + 14
  lInSize = 2
  lOutSize = 1
  USB_IO
End Sub

Function RdPort(Port) As Integer
  lIn = Port * 256 + 20
  lInSize = 2
  lOutSize = 2
  USB_IO
  RdPort = (lOut / 256) And 255
End Function

Sub WrRAM(Adresse, Wert)
  lIn = 65536 * Wert + Adresse * 256 + 23
  lInSize = 3
  lOutSize = 1
  USB_IO
End Sub

Private Sub HScroll1_Change()
  Wert = HScroll1.Value * 8 + 7
  WrRAM 46, Wert
  Label4.Caption = Str$(Wert)
End Sub

Private Sub HScroll2_Change()
  WrRAM 47, HScroll2.Value
  Label5.Caption = HScroll2.Value
End Sub

Private Sub HScroll3_Change()
  Brightness HScroll3.Value
  Label6 = HScroll3.Value
End Sub

Private Sub Timer1_Timer()
  lIn = 11
  lInSize = 1
  lOutSize = 3
  USB_IO
  Temp = ((lOut \ 256) And 255) / 2
  Minus = (lOut \ 65536) And 255
  If Minus > 0 Then Temp = Temp * -1
  Button = (lOut \ 16777216) And 255
  Label7.Caption = Str$(Temp)
  Label8.Caption = Str$(Button)
```

```
  Label9.Caption = Str$(RdPort(0))
  Label10.Caption = Str$((RdPort(1)) And 15)
End Sub
```

Listing 5.9: Alle Zugriffe auf die Ports in USBPORTS.FRM

6 Ein universelles USB-Interface

Hier soll eine konkrete USB-Entwicklung vorgestellt werden. Die Aufgabe bestand darin, ein existierendes Interface an der RS232 (CompuLAB der Firma Modul-Bus) für den USB neu zu entwickeln. Die wichtigsten Schritte dieser Entwicklung werden hier dokumentiert und können als Grundlage eigener Versuche dienen. Das fertige Gerät wird von Modul-Bus unter der Bezeichnung CompuLAB-USB gebaut. Ein Eigenbau ist möglich, da alle erforderlichen Informationen offenliegen.

Die Daten des alten CompuLAB sollten eingehalten oder verbessert werden. Daraus ergeben sich folgende Mindestforderungen an das neu zu entwickelnde Interface:

8 Digitale Ausgänge, TTL-kompatibel

8 Digitale Eingänge, CMOS-kompatibel, hochohmig

2 analoge Eingänge, Auflösung 8 Bit, Bereich 0...5 V

Da der CY7C63000 anders als z.B. der ST6-Controller des Vorgängers keinen internen AD-Wandler besitzt, muss zunächst ein geeigneter AD-Wandler ausgesucht werden. Ein kostengünstiger Typ ist der TLC541 mit 11 Eingängen und einer Auflösung von 8 Bit. Gegen diesen Wandler spricht aber, dass er einen externen Takt unter 2 MHz benötigt. Der nächstgrößere Wandler von Texas Instruments ist der TLC1543 mit einer Auflösung von 10 Bit. Er besitzt eine interne Takterzeugung und benötigt zur Ansteuerung vier oder fünf Portleitungen, je nachdem ob das End-Of-Conversion Signal ausgewertet wird oder man eine einfache Zeitschleife verwendet. Auf diesen Wandler fiel die Wahl auch deshalb, weil er ein günstiges Preis/Leistungsverhältnis besitzt.

6.1 Der AD-Wandler TLC1543

Der AD-Wandler TLC1543 ist ein CMOS-Baustein mit seriell getaktetem Interface zur direkten Ansteuerung durch Mikrocontroller. Die digitalen Daten erscheinen also an einer einzelnen Leitung und werden nacheinander vom Mikrocontroller eingelesen. Eine Taktleitung steuert die Datenübertragung wie bei einem Schieberegister. In gleicher Weise werden Daten, hier speziell die Adresse des Eingangskanals, vom Mikrocontroller in den Wandler übertragen.

Die Betriebsspannung ist 5 V, die typische Stromaufnahme ca. 1 mA. Der Wandler arbeitet nach dem Verfahren der sukzessiven Approximation mit einem Netzwerk aus binär gewichteten Kondensatoren. Abb. 6.1 zeigt das Prinzipschaltbild.

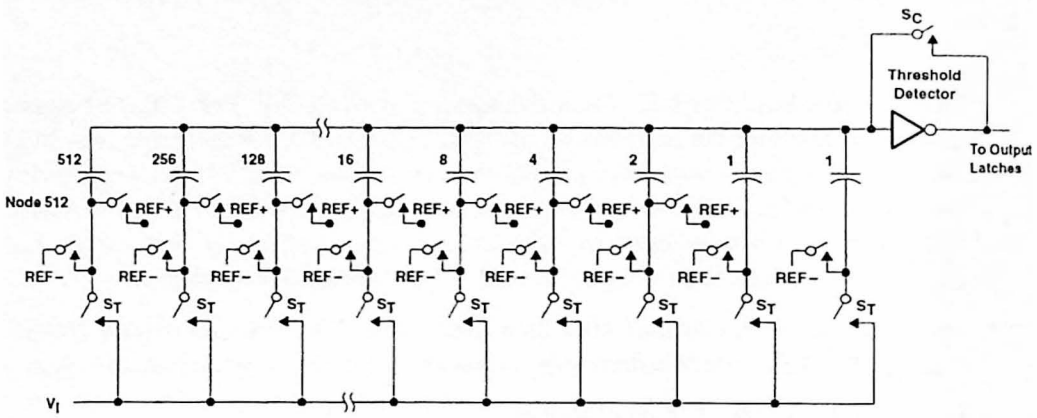


Abb. 6.1: Prinzip des Wandlers mit geschalteten Kondensatoren (Texas Instruments)

Der CMOS-Komparator prüft jedes Bit durch einen Ladungsvergleich. In der ersten Phase der Wandlung wird ein Analogwert aufgenommen, indem der Schalter S-C und alle Schalter S-T geschlossen werden. Alle Kondensatoren werden dabei auf die Eingangsspannung aufgeladen, wobei ihr gemeinsamer Anschluss auf der Schaltschwelle des Komparators bei etwa der Hälfte der Betriebsspannung liegt. Dann werden die Schalter geöffnet. Der Komparator prüft als nächstes die Ladung jedes Kondensators. Zuerst wird der Knoten 512 an REF+ gelegt und alle anderen Knoten an REF-. Wenn die resultierende Spannung unter der Schaltschwelle liegt, wurde für Bit 9 ein Eins-Bit erkannt, und der Kondensator bleibt an REF+. Der gleiche Prozess wird dann mit dem Kondensator der Wertigkeit 256 wiederholt, um das Bit 8 zu gewinnen. Bei jedem Schritt der sukzessiven Approximation wird die ursprüngliche Ladung auf alle Kondensatoren verteilt. Nach insgesamt 10 Vergleichen ist die Spannung auf zehn Bit genau bestimmt.

Der Eingang des Wandlers wird durch einen internen Multiplexer jeweils an einen von 11 externen Eingängen gelegt. Zusätzlich gibt es drei interne Kanäle, die einen Selbsttest ermöglichen. Kanal A11 liegt intern an der halben Referenzspannung und liefert ein Ergebnis von 512. Mögliche Abweichungen geben wertvolle Hinweise auf Störungen wie z.B. unzureichend gesiebte Betriebsspannung.

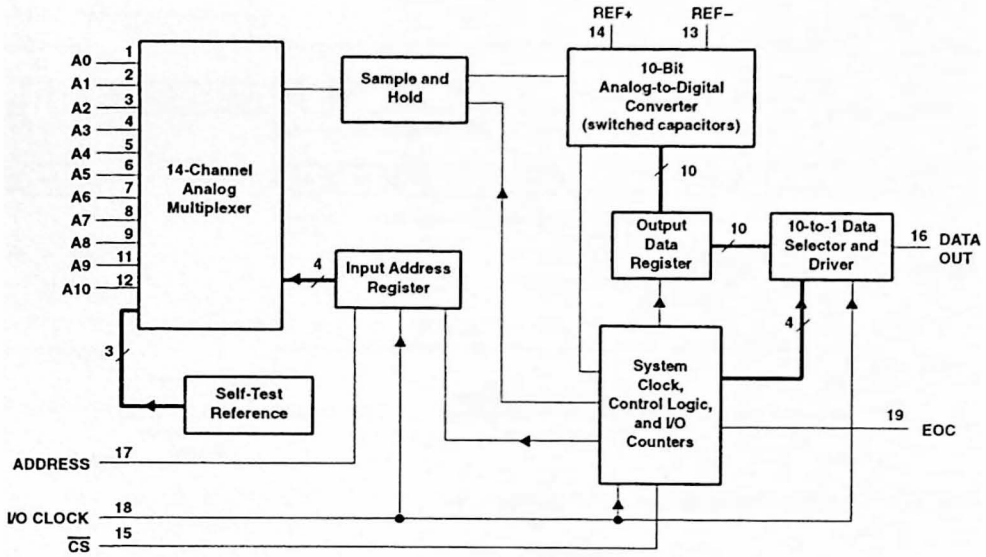


Abb. 6.2: Blockschaltbild des TLC1543

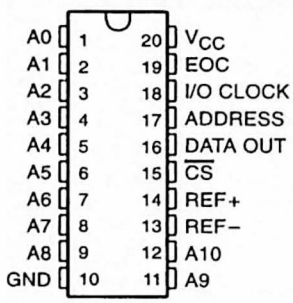


Abb. 6.3: Die Pinbelegung des TLC1543

Der TLC1543 verwendet eine Chip-Select-Leitung /CS zur Aktivierung der Datenübertragung mit dem Controller. Die A/D-Wandlung wird ausgeführt, solange /CS hoch liegt, also inaktiv ist. Jede getaktete Datenübertragung liefert gleichzeitig 10 Datenbits der letzen Wandlung an den Mikrocontroller und die Kanaladresse der nächsten Wandlung an den AD-Wandler.

Der Wandler soll in einem ersten Versuch mit möglichst geringen Änderungen der Software an das Starterboard angeschlossen werden. Es bietet sich an, die Thermometerroutine des Beispielprogramms durch eine AD-Routine zu ersetzen. Entsprechend werden die Portanschlüsse P0.0 bis P0.4 verwendet.

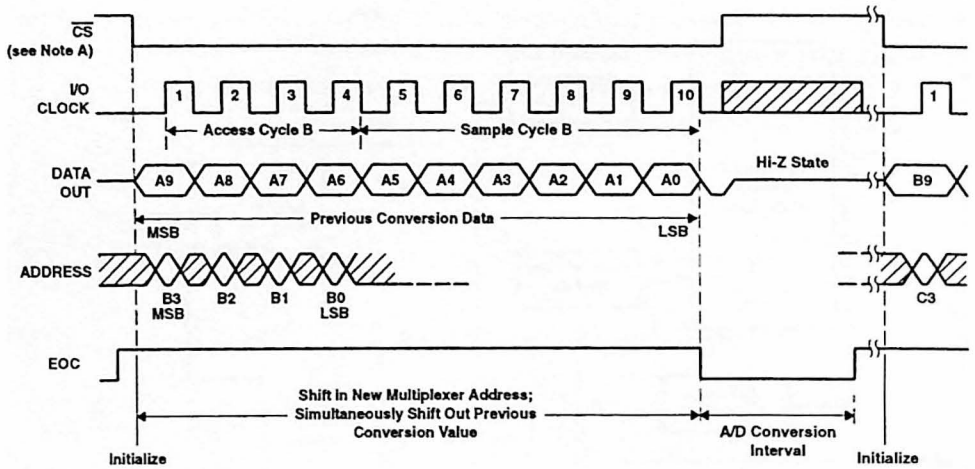
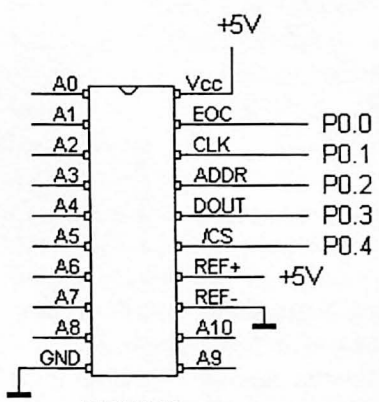


Abb. 6.4: Das Impulsdiagramm der Ansteuerung

Der direkte Ersatz der Thermometer-Routine führt ohne Änderungen am Kern des Controller-Programms dazu, dass die AD-Routine TLC1543a.asm, über einen Timer gesteuert, alle 1024 µs aufgerufen wird. Die Ergebnisse der Wandlung werden in den Speicherzellen 78h und 79h übergeben.



TLC1543

Abb. 6.5: Anschluss des Wandlers an Port 0

```
; TLC1543a.asm - 10 Bit 12 Chanell AD-Converter
ADPort                  :equ 00h     ; SysPort0
ADMaskBits              :equ 1fh     ;
ADEOC                   :equ 01h     ; P0.0
ADClock                 :equ 02h     ; P0.1
ADDataIn                :equ 04h     ; P0.2
```

```
ADDataOut                   :equ 08h      ; P0.3
ADCS                        :equ 10h      ; P0.4

gbADChan                    :equ 31h      ;
gbADPortMirror              :equ 32h      ;
gbADBitcount                :equ 33h      ;
gbADChanIn                  :equ 34h      ;
gbADDatHi                   :equ 78h      ;USBEndP1FIFO
gbADDatLo                   :equ 79h      ;USBEndP1FIFO +1

;//$PAGE
;***********************************************
; ADRead()
;
;***********************************************

  ADRead:
    mov  a,[gbADChanIn]             ;read analog input
    mov  [gbADChan],a
    mov  a,[gbPort0]                ; setup port bits
    and  a,~ADClock
    or   a,(ADDataOut|ADCS|ADEOC)
    iowr ADPort
    mov  [gbADPortMirror],a
    mov  a,00h
    mov  [gbADDatHi],a
    mov  [gbADDatLo],a
    mov  a,[gbADPortMirror]         ;CS low
    and  a,~ADCS
    iowr ADPort
    mov  [gbADPortMirror],a
    mov  a,8                        ;8 bits count
  ADLoop1:
    mov  [gbADBitcount],a
    mov  a,[gbADDatHi]              ;output data shift left
    asl  a
    mov  [gbADDatHi],a
    iord ADPort                     ;read one Bit from Dout
    and  a,ADDataOut
    cmp  a,ADDataOut
    jnz  Ainlow1
    mov  a,[gbADDatHi]
    or   a,01h
    mov  [gbADDatHi],a
  Ainlow1:
    mov  a,[gbADPortMirror]         ;output one bit to Din
    and  a,~ADDataIn
    mov  [gbADPortMirror],a
    mov  a,[gbADChan]
```

```
  and  a,80h
  cmp  a,80h
  jnz  AoutLow
  mov  a,[gbADPortMirror]
  or   a,ADDataIn
  mov  [gbADPortMirror],a
AoutLow:
  mov  a,[gbADPortMirror]
  iowr ADPort
  or   a,ADClock                ; set clock high
  iowr ADPort
  mov  a,[gbADChan]             ;input data shift left
  asl  a
  mov  [gbADChan],a
  nop
  nop
  nop
  nop
  mov  a,[gbADPortMirror]
  and  a,~ADClock               ; set clock low
  iowr ADPort
  mov  [gbADPortMirror],a
  mov  a,[gbADBitcount]         ; complete loop 1
  dec  a
  jnz  ADLoop1
                                ; read the two bytes left
  mov  a,02h
 ADLoop2:
  mov  [gbADBitcount],a
  mov  a,[gbADDatLo]            ;output data shift left
  asl  a
  mov  [gbADDatLo],a
  iord ADPort                   ;read one Bit from Dout
  and  a,ADDataOut
  cmp  a,ADDataOut
  jnz  Ainlow2
  mov  a,[gbADDatLo]
  or   a,01h
  mov  [gbADDatLo],a
Ainlow2:
  mov  a,[gbADPortMirror]       ;set clock high
  or   a,ADClock
  iowr ADPort
  nop
  nop
  nop
  nop
  and  a,~ADClock               ;set clock low
  iowr ADPort
```

```
    mov  [gbADPortMirror],a
    mov  a,[gbADBitcount]          ;complete loop 2
    dec  a
    jnz  ADLoop2
    mov  a,[gbADPortMirror]        ;set CS high
    or   a,ADCS
    iowr ADPort
    ret
```

Listing 6.1: Ansteuerung des TLC1541 über Port 0

Das Programm verwendet zwei Schleifen mit insgesamt 12 Taktimpulsen. In der ersten Schleife werden acht Datenbits gelesen und acht Datenbits in den Wandler geschoben. Das Hauptprogramm übergibt im globalen Byte gbADChanIn den gewünschten Analogeingang. Da die ersten vier übertragenen Bits die Adresse für den internen Multiplexer des AD-Wandlers darstellen, sind hier Vielfache von 10h zu übergeben (00, 16, 32, 48 ...). Die Routine kopiert die Kanalnummer in die Variable gbADChan, die in der Schleife acht mal nach links verschoben wird, um jeweils ein Bit zu isolieren. In ähnlicher Weise werden nacheinander die ersten acht Bits der letzten Wandlung ausgelesen und in gbADDatHi geschoben.

Eine zweite Schleife verarbeitet die beiden niederwertigen Bits der 10-Bit-Messung. Hier ist keine Ausgabe an den Wandler mehr erforderlich. Das Lowbyte wird in der Speicheradresse 79h im Bereich des nicht verwendeten Endpoint-1-FIFO abgelegt. Der Anwender kann je nach Anforderung nur das Highbyte in 78h lesen und erhält dann eine Genauigkeit von 8 Bit, oder er verwendet zusätzlich das Lowbyte, um die Genauigkeit auf 10 Bit zu steigern. Die gesamte AD-Routine benötigt ca. 140 µs. Es ist nicht erforderlich, eine Conversion-Zeit zu berücksichtigen, da Wandlung nur einmal pro Millisekunde gestartet wird. Damit stehen also 860 µs zur Verfügung, während der Wandler mit 21 µs auskommt. Andererseits wurden NOP-Befehle in die Schleife eingefügt, um eine ausreichend lange High-Phase der Taktimpulse zu gewährleisten. Die relativ großen Pullups und der sehr kleine Ausgangsstrom am Port 1 führten zusammen mit den unvermeidlichen Kapazitäten zu einem Verschleifen der Taktimpulse, das nur mit einer Verlängerung der Taktimpulse tolerierbar erschien.

In diesem ersten Versuch wurden die Daten direkt im Ausgabebereich 78/79h aufbereitet. Es zeigte sich jedoch, dass ein gewisser Prozentsatz der Daten gestört wurde. Als Ursache kam nur die Interruptverwaltung in Frage. Die AD-Routine selbst läuft im 1024-µs-Timerinterrupt. Sie kann unterbrochen werden, wenn eine Endpoint-0-Ereignis auftritt. Die AD-Routine benötigt ca. 140 µs. Damit besteht eine Wahrscheinlichkeit von etwa 14%, dass ein Lesezugriff auf das Ergebnis gerade erfolgt, wenn die Wandlung noch läuft.

Eine Verbesserung ergibt sich, wenn die Daten in zwei anderen Speicherstellen aufbereitet werden und erst am Ende in die Ausgabestelle kopiert werden.

Ein anderes Problem ist das Zeitverhalten. Ein Control-Zugriff benötigt jeweils drei USB-Frames, also 3 ms. Hier werden für jeden Kanal ein RAM-Schreibzugriff und zwei RAM-Lesezugriffe benötigt. Damit kommt man auf einen Zeitbedarf von neun Millisekunden pro Kanal. Dem steht eine Millisekunde gegenüber, die beim RS232-Vorgänger des Interfaces benötigt wurde. Es gab einen Abfragebefehl pro Kanal. Dazu wurde ein Byte abgesandt und ein Antwortbyte empfangen. Bei 19200 Baud ergab sich daher eine Zeit von ca. 1 ms.

6.2 Zusammenfassung von AD-Kanälen

Der Nachteil einer langsameren Abfrage des USB-Geräts kann ausgeglichen werden, wenn mehrere Kanäle gleichzeitig ausgelesen werden. Bis zu acht Bytes können in einem Request gemeinsam übertragen werden. Damit ist es möglich, vier Kanäle zusammen abzufragen. Die Übertragung der Kanalnummer kann entfallen, wenn die Firmware automatisch den Kanal inkrementiert und mehrere Messungen zusammen ausführt. Die eigentliche Messung an vier Kanälen kann im Timer-Interrupt ablaufen, wenn sie deutlich weniger als eine Millisekunde benötigt. Die Messdaten für vier Kanäle werden einmal pro Millisekunde erneuert. Die Abfrage von vier Kanälen benötigt nur drei Millisekunden, wenn eine entsprechende Treiberfunktion zur Abfrage von acht Bytes zur Verfügung steht. Damit ergibt sich ein kleiner Geschwindigkeitsvorteil gegenüber der RS232-Lösung.

Die endgültige Portnutzung sollte mit einem Minimum an Bauteilen auskommen. Da der AD-Wandler mit 11 Eingangskanälen mehr bietet als benötigt wird, liegt die Überlegung nahe, acht der analogen Eingänge als Digitaleingänge zu nutzen. Dies erfordert eine Messung an jedem der acht Kanäle und eine Gewichtung des Messergebnisses. Im Endeffekt ist es auf diese Weise möglich, Port 0 des Mikrocontrollers ganz für die digitalen Ausgaben zu nutzen, während Port 1 mit seinen vier Leitungen den AD-Wandler bedient. Die EOC-Leitung wird nicht ausgewertet, so dass vier Portleitungen ausreichen. Bei einer geeigneten Zuordnung kann auch die Bereitschafts-LED mit angesteuert werden.

Die Schaltung nach Abb. 6.6 verwendet die ersten acht Eingangskanäle als Digitaleingänge. Damit stehen drei Eingänge als Analogeingänge zur Verfügung. Der zusätzliche Eingang stellt eine Verbesserung gegenüber dem ursprünglichen Interface (CompuLAB-RS232) dar. Bei geeigneter Auslegung der Eingangsbeschaltung kann überdies erreicht werden, dass alle Digitaleingänge wahlweise auch als Analogeingänge verwendet werden können. Bei Bedarf hätte man also alle 11 Analogeingänge.

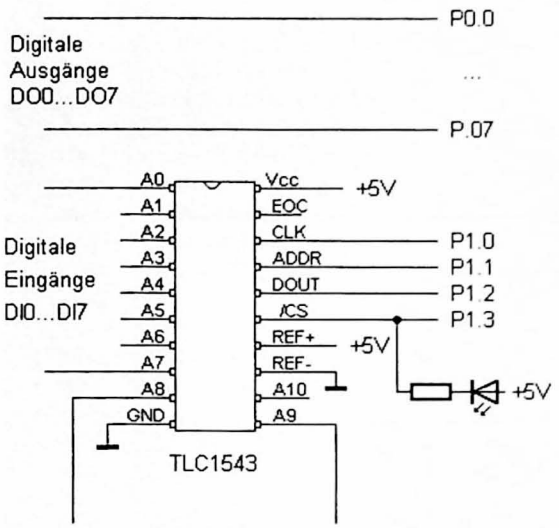


Abb. 6.6: Zuordnung der Ports

Aus Zeitgründen sollen vier AD-Kanäle zusammen gemessen werden. Der Anwender kann einen Startkanal festlegen, von dem aus vier Kanäle gezählt werden. Die Wandlerroutine muss dazu fünfmal aufgerufen werden, weil die gewandelten Daten immer erst beim folgenden Aufruf ausgelesen werden können. Zwischen den einzelnen seriellen Zugriffen auf den Wandler muss diesmal eine Wandlungszeit von 21 µs eingehalten werden. Dafür können aber NOP-Befehle zur Verlängerung der Taktimpulse eingespart werden, weil die kräftigeren Port-1-Ausgänge steilere Impulsflanken liefern. Insgesamt wird die Wandlung nun in 650 µs ausgeführt. Damit ist man noch weit genug von der Grenze 1024 µs entfernt. Die verbleibenden 350 µs müssen für andere Interrupt-Ereignisse ausreichen, insbesondere für die USB-Requests.

```
; TLC1543b.asm - 10 Bit 12 Chanell AD-Converter

ADPort                  :equ 01h     ; SysPort1
ADMaskBits              :equ 1fh     ;
ADClock                 :equ 01h     ; P1.0
ADDataIn                :equ 02h     ; P11
ADDataOut               :equ 04h     ; P12
ADCS                    :equ 08h     ; P1.3

gbADChan                :equ 31h     ;
gbADPortMirror          :equ 32h     ;
gbADBitcount            :equ 33h     ;
gbADChanIn              :equ 34h     ;
```

```
gbADDatHi                   :equ 35h     ;
gbADDatLo                   :equ 36h     ;

;//$PAGE
;************************************************************
; ADRead()  Read gBADChan/+1/+2/+3
;
;************************************************************
  ADRead:
    mov  a,[gbADChanIn] ;read analog input base
    add  a,48                              ;Base + 3
    mov  [gbADChan],a
    call ADC

    mov  a,[gbADChanIn]       ;read analog input
    add  a,32                 ;convert (base+2)
    mov  [gbADChan],a
    call ADC
    mov  a,[gbADDatHi]        ;result (base +3)
    mov  [7Eh],a
    mov  a,[gbADDatLo]
    mov  [7Fh],a

    mov  a,[gbADChanIn]       ;read analog input
    add  a,16                 ;convert (base+1)
    mov  [gbADChan],a
    call ADC
    mov  a,[gbADDatHi]        ;result (base +2)
    mov  [7Ch],a
    mov  a,[gbADDatLo]
    mov  [7Dh],a

    mov  a,[gbADChanIn]       ;read analog input
    mov  [gbADChan],a         ;convert (base)
    call ADC
    mov  a,[gbADDatHi]        ;result (base +1)
    mov  [7Ah],a
    mov  a,[gbADDatLo]
    mov  [7Bh],a

    mov  a,[gbADChanIn]       ;read analog input
    mov  [gbADChan],a         ;convert (base)
    call ADC
    mov  a,[gbADDatHi]        ;result (base)
    mov  [78h],a
    mov  a,[gbADDatLo]
    mov  [79h],a
    ret
```

```
ADC:
  mov  a,[gbPort0]          ; setup port bits
  and  a,~ADClock
  or   a,(ADDataOut|ADCS)
  iowr ADPort
  mov  [gbADPortMirror],a
  mov  a,00h
  mov  [gbADDatHi],a
  mov  [gbADDatLo],a
  mov  a,[gbADPortMirror]   ;CS low
  and  a,~ADCS
  iowr ADPort
  mov  [gbADPortMirror],a
  mov  a,8                  ;8 bits count
ADLoop1:
  mov  [gbADBitcount],a
  mov  a,[gbADDatHi]        ;output data shift left
  asl  a
  mov  [gbADDatHi],a
  iord ADPort               ;read one Bit from Dout
  and  a,ADDataOut
  cmp  a,ADDataOut
  jnz  Ainlow1
  mov  a,[gbADDatHi]
  or   a,01h
  mov  [gbADDatHi],a
Ainlow1:
  mov  a,[gbADPortMirror]   ;output one bit to Din
  and  a,~ADDataIn
  mov  [gbADPortMirror],a
  mov  a,[gbADChan]
  and  a,80h
  cmp  a,80h
  jnz  AoutLow
  mov  a,[gbADPortMirror]
  or   a,ADDataIn
  mov  [gbADPortMirror],a
AoutLow:
  mov  a,[gbADPortMirror]
  iowr ADPort
  or   a,ADClock            ; set clock high
  iowr ADPort
  mov  a,[gbADChan]         ;input data shift left
  asl  a
  mov  [gbADChan],a
  mov  a,[gbADPortMirror]
  and  a,~ADClock           ; set clock low
  iowr ADPort
  mov  [gbADPortMirror],a
```

```
  mov  a,[gbADBitcount]     ; complete loop 1
  dec  a
  jnz  ADLoop1
                            ; read the two bytes left
  mov  a,02h
 ADLoop2:
  mov  [gbADBitcount],a
  mov  a,[gbADDatLo]        ;output data shift left
  asl  a
  mov  [gbADDatLo],a
  iord ADPort               ;read one Bit from Dout
  and  a,ADDataOut
  cmp  a,ADDataOut
  jnz  Ainlow2
  mov  a,[gbADDatLo]
  or   a,01h
  mov  [gbADDatLo],a
Ainlow2:
  mov  a,[gbADPortMirror]   ;set clock high
  or   a,ADClock
  iowr ADPort
  nop
  nop
  and  a,~ADClock           ;set clock low
  iowr ADPort
  mov  [gbADPortMirror],a
  mov  a,[gbADBitcount]     ;complete loop 2
  dec  a
  jnz  ADLoop2
  mov  a,[gbADPortMirror]   ;set CS high
  or   a,ADCS
  iowr ADPort
  nop
  nop
  nop
  nop
  nop
  nop  ;4us

  nop
  nop
  nop
  nop
  nop
  nop  ;4us

  nop
  nop
  nop
```

```
    nop
    nop
    nop  ;4us

    nop
    nop
    nop
    nop
    nop
    nop  ;4us

    nop
    nop
    nop
    nop
    nop
    nop  ;4us
    ret
```

Listing 6.2: Automatisches Erfassen von vier Analogkanälen

Der Startkanal wird auf Kanal 8 eingestellt. Die Abfrage über RdRAM-Requests liefert also im Normalfall Ergebnisse für die Kanäle 8 bis 11. Bei Bedarf kann jedoch über die RAM-Adresse 34h ein anderer Anfangskanal festgelegt werden. Auf diese Weise kann ein Anwenderprogramm auch auf die Kanäle 0 bis 7 zugreifen.

6.3 AD-Eingänge als Digitaleingänge

Die Verwendung der ersten acht Analogeingänge als Digitaleingänge bringt gewisse Zeitprobleme mit sich. Nach Möglichkeit sollte eine gleichzeitige Änderung mehrerer Eingangsbits auch gleichzeitig gelesen werden können. Mit der bisherigen Messroutine können aber nicht alle acht Kanäle in einer Millisekunde abgefragt werden. Zusätzlich ist eine Bewertung der Ergebnisse erforderlich, wobei Messwerte oberhalb einer festgelegten Schwelle als logische Einsen gelten. Und das Gesamtergebnis muss zu einem Byte verarbeitet werden. Damit all diese Vorgänge für alle acht Eingänge in deutlich weniger als einer Millisekunde ausgeführt werden können, ist eine spezielle Anpassung der Wandlerroutine nötig.

Zunächst reicht es, das höchstwertige Bit des Wandler-Ergebnisses auszulesen. Die Schaltschwelle liegt damit bei 50% der Referenzspannung, also bei 2,5 V. Das Ergebnis der letzten Wandlung erscheint bereits vor dem ersten Taktimpuls, sobald CS heruntergezogen wird. Das Lesen des einen Bits kann daher vor der eigentlichen Schleife mit Taktimpulsen erfolgen. Es wird Zeit eingespart, weil der Lesevorgang

nur noch einmal pro Kanal ausgeführt wird. Eine weitere Einsparung bringt eine Umorganisation der beiden Schleifen. Die erste Schleife mit vier Taktimpulsen führt die Übertragung der Kanaladresse aus. Die zweite Schleife erzeugt nur noch die restlichen acht Taktimpulse, die erforderlich sind, damit die eigentliche Wandlung gestartet wird. Auch die Wandlungszeit selbst kann abgekürzt werden, da nur das MSB gelesen werden soll, das intern zuerst gebildet wird.

Die verkürzte Wandlerroutine muss insgesamt neunmal aufgerufen werden, um alle acht Eingangsbits zu lesen. Durch Schiebebefehle wird das Gesamtergebnis in einem Byte gebildet. Aus Gründen der Zeitersparnis wird hier auf eine zusätzliche Schleife verzichtet und stattdessen „straight down" programmiert. Der Lesevorgang beginnt mit Kanal 7 und endet mit Kanal 0. Für die gesamte Routine wurde ein Zeitbedarf von ca. 700 µs gemessen.

```
RdDin:
  mov A,7
  mov [gbADChan],a
  call ADC2                 ;convert ch7

  mov A,0
  mov [gbADDatHi],a         ;clear data
  mov A,6
  mov  [gbADChan],a
  call ADC2                 ;read ch7
  mov  a,[gbADDatHi]
  asl  a
  mov  [gbADDatHi],a

  mov A,5
  mov  [gbADChan],a
  call ADC2                 ;read ch6
  mov  a,[gbADDatHi]
  asl  a
  mov  [gbADDatHi],a

  mov A,4
  mov  [gbADChan],a
  call ADC2                 ;read ch5
  mov  a,[gbADDatHi]
  asl  a
  mov  [gbADDatHi],a

  mov A,3
  mov  [gbADChan],a
  call ADC2                 ;read ch4
  mov  a,[gbADDatHi]
  asl  a
  mov  [gbADDatHi],a
```

```
  mov A,2
  mov  [gbADChan],a
  call ADC2                  ;read ch3
  mov  a,[gbADDatHi]
  asl  a
  mov  [gbADDatHi],a

  mov A,1
  mov  [gbADChan],a
  call ADC2                  ;read ch2
  mov  a,[gbADDatHi]
  asl  a
  mov  [gbADDatHi],a

  mov A,0
  mov  [gbADChan],a
  call ADC2                  ;read ch1
  mov  a,[gbADDatHi]
  asl  a
  mov  [gbADDatHi],a

  mov A,0
  mov  [gbADChan],a
  call ADC2                  ;read ch0
  mov  a,[gbADDatHi]
  mov  [gbDIn],a
  ret

 ADC2:
  mov  a,[gbPort0]           ; setup port bits
  and  a,~ADClock
  or   a,(ADDataOut|ADCS)
  iowr ADPort
  mov  [gbADPortMirror],a
  mov  a,[gbADPortMirror]    ;CS low
  and  a,~ADCS
  iowr ADPort
  mov  [gbADPortMirror],a

  iord ADPort                 ;read MSB Dout
  and  a,ADDataOut
  cmp  a,ADDataOut
  jnz  Ainlow3
  mov  a,[gbADDatHi]
  or   a,01h
  mov  [gbADDatHi],a
Ainlow3:
```

```
  mov  a,4                  ;4 bits count
ADLoop3:
  mov  [gbADBitcount],a
  mov  a,[gbADPortMirror]   ;output one bit to Din
  and  a,~ADDataIn
  mov  [gbADPortMirror],a
  mov  a,[gbADChan]
  and  a,08h
  cmp  a,08h
  jnz  AoutLow2
  mov  a,[gbADPortMirror]
  or   a,ADDataIn
  mov  [gbADPortMirror],a
AoutLow2:
  mov  a,[gbADPortMirror]
  iowr ADPort
  or   a,ADClock            ; set clock high
  iowr ADPort
  mov  a,[gbADChan]         ;input data shift left
  asl  a
  mov  [gbADChan],a
  mov  a,[gbADPortMirror]
  and  a,~ADClock           ; set clock low
  iowr ADPort
  mov  [gbADPortMirror],a
  mov  a,[gbADBitcount]     ; complete loop 1
  dec  a
  jnz  ADLoop3
                            ; read the two bytes left
  mov  a,08h
ADLoop4:
  mov  [gbADBitcount],a
  mov  a,[gbADPortMirror]   ;set clock high
  or   a,ADClock
  iowr ADPort
  nop
  nop
  and  a,~ADClock           ;set clock low
  iowr ADPort
  mov  [gbADPortMirror],a
  mov  a,[gbADBitcount]     ;complete loop 2
  dec  a
  jnz  ADLoop4
  mov  a,[gbADPortMirror]   ;set CS high
  or   a,ADCS
  iowr ADPort
  nop
  nop
  nop
```

```
    nop
    nop
    nop  ;4us

    ret
```

Listing 6.3: Lesen von acht Datenbits der analogen Eingänge 0..7

Die nächste Aufgabe besteht darin, analoge und digitale Eingaben zu verbinden. Sie können nicht in einem Timerinterrupt zusammengefasst werden, weil sie zusammen mehr als eine Millisekunde benötigen. Es erscheint daher sinnvoll, die Messungen zu toggeln, d.h. in einer Millisekunde werden die digitalen Eingänge erfasst, in der nächsten die analogen. Nach jeweils zwei Millisekunden werden alle Eingänge neu erfasst. Da ein Lesevorgang mindestens drei Millisekunden dauert, findet der Anwender im Rahmen der zeitlichen Auflösung immer aktuelle Daten.

Zusätzlich sollen die Daten so zusammengefasst werden, dass sie gemeinsam in einem USB-Request abgeholt werden können. Da die FIFO-Größe auf acht Bytes begrenzt ist, müssen die Lowbytes der anlogen Ergebnisse zusammengefasst werden. Da sie jeweils nur zwei Bits benötigen, könnte man alle vier Lowbytes in einem Byte unterbringen. Man kommt jedoch auch mit zwei Lowbytes pro Speicherplatz aus und hat dann noch den Vorteil einer späteren Erweiterbarkeit auf eine Auflösung von 12 Bits. Es wird daher folgende Aufteilung gewählt:

79h Digital-Eingänge
7Ah Kanal 8 Highbyte
7Bh Kanal 9 Highbyte
7Ch Kanal 8..9, Lowbytes
7Dh Kanal 10 Highbyte
7Eh Kanal 11 Highbyte
7Fh Kanal 10,11, Lowbytes

Die gewonnenen Daten können weiterhin mit dem Treiber USBTherm.sys gelesen werden, wobei das Read-RAM-Kommando zum Auslesen einzelner Bytes verwendet wird. Das Programm-Modul TLC1543e.asm enthält die Messung und das Zusammenfügen der Ergebnisbytes.

```
; TLC1543e.asm - 10 Bit 12 Chanell AD-Converter

ADPort                  :equ 01h     ; SysPort1
ADMaskBits              :equ 1fh     ;

ADClock                 :equ 01h     ; P1.0
ADDataIn                :equ 02h     ; P1.1
```

```
ADDataOut                   :equ 04h      ; P1.2
ADCS                        :equ 08h      ; P1.3

gbADChan                    :equ 31h      ;
gbADPortMirror              :equ 32h      ;
gbADBitcount                :equ 33h      ;
gbADChanIn                  :equ 34h      ;
gbADDatHi                   :equ 35h      ;
gbADDatLo                   :equ 36h      ;
gbDIn          :equ 37h
gbToggle                    :equ 38h

;*************************************************************
; ADRead()  Toggle analog and digital inputs
;*************************************************************

  ADDRead:
    mov  a,[gbToggle]
    cmp  a,1
    jnz  Digi
  Ann:
    call ADRead
    mov  a,0
    mov  [gbToggle],a
    ret
  Digi:
    call RdDin
    mov  a,1
    mov  [gbToggle],a
    ret

;*****************************************************
; ADRead()  Read 4 inputs gBADChan/+1/+2/+3
;*****************************************************

  ADRead:
    mov  a,[gbADChanIn]       ;read analog input base
    add  a,48                 ;Base + 3
    mov  [gbADChan],a
    call ADC

    mov  a,[gbADChanIn]       ;read analog input
    add  a,32                 ;convert (base+2)
    mov  [gbADChan],a
    call ADC
    mov  a,[gbADDatHi]        ;result (base +3)
    mov  [7Eh],a
    mov  a,[gbADDatLo]
    mov  [7Fh],a
```

```
mov  a,[gbADChanIn]       ;read analog input
add  a,16                 ;convert (base+1)
mov  [gbADChan],a
call ADC
mov  a,[gbADDatHi]        ;result (base +2)
mov  [7Dh],a
mov a,[7Fh]
asl  a
asl  a
asl  a
asl  a
add  a,[gbADDatLo]
mov  [7Fh],a

mov  a,[gbADChanIn]       ;read analog input
mov  [gbADChan],a         ;convert (base)
call ADC
mov  a,[gbADDatHi]        ;result (base +1)
mov  [7Bh],a
mov  a,[gbADDatLo]
mov  [7Ch],a
mov  a,[gbADChanIn]       ;read analog input
mov  [gbADChan],a         ;convert (base)
call ADC
mov  a,[gbADDatHi]        ;result (base)
mov  [7Ah],a
mov a,[7Ch]
asl  a
asl  a
asl  a
asl  a
add  a,[gbADDatLo]
mov  [7Ch],a
ret
```

Listing 6.4: Umverteilung von je zwei Lowbytes auf ein Datenbyte

Abb. 6.7 zeigt das Zeitverhalten des Programms. Die Messung wurde mit dem in Kap. 10 vorgestellten Speicher-Scope durchgeführt. Gesteuert vom Timer-Interrupt werden in einer Millsekunde alle Analogmessungen ausgeführt, in der anderen alle verkürzten Messungen an den digitalen Eingangskanälen. Jeweils die erste Messung muss doppelt ausgeführt werden, weil zuerst nur die Adresse des Eingangskanals übertragen wird, dessen Wert erst im zweiten Zugriff ausgelesen werden kann.

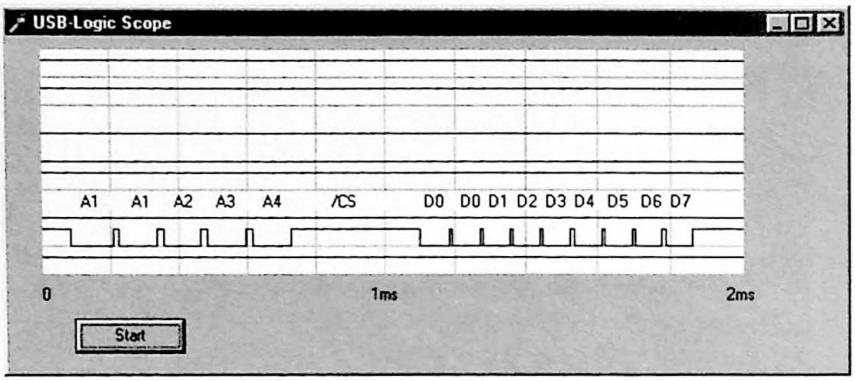


Abb. 6.7: Das Zeitverhalten der AD-Routine

6.4 Gemeinsame Übertragung der Nutzdaten

In einem nächsten Schritt kann man versuchen, auch die digitalen Ausgänge mit dem gleichen Request zu setzen. Wenn man davon ausgeht, dass ein Anwenderprogramm die volle Kontrolle über alle Ein- und Ausgänge benötigt, ist es sinnvoll, mit einem USB-Request alle I/O-Funktionen zu erneuern. Das Anwenderprogramm sendet also aktuelle Zustände der digitalen Ausgänge und erhält die Zustände aller Eingänge zurück.

Dies ist jedoch nur mit einem neuen Treiber zu realisieren, weil der ursprüngliche Treiber keine Übertragung von acht Bytes in einem Request ermöglicht. Eine Verbesserung bringt der Thermometer-Treiber von Craig Peacock, der weiter unten in Kapitel 7 genauer vorgestellt wird. Dieser Treiber verwendet zusätzliche Control-Codes. Mit IOCtrlCode=8 ist es möglich, die ganze Länge eines FIFOs in einem Request zu übertragen. Man kann also nun neue Funktionen mit langen Datenblöcken in die Firmware einbauen.

Die bisherige Firmware verwendet intern die Vendor-Requestfunktionen 1 bis 5. Es lässt sich leicht eine zusätzliche Funktion 6 unterbringen, die das gewünschte Verhalten zeigt. An der FIFO-Position 1 soll das Datenbyte für die digitalen Ausgänge empfangen werden. Anschließend wird der gesamte Datenbereich des Endpoint 1 in den Bereich des Endpoint 0 kopiert. Beim Absenden der acht Bytes wird die Position 0 mit dem ACK-Byte 42h gefüllt. Es bleiben noch sieben reine Nutzdaten, also gerade genug für die Aufgabe.

```
USBEventEP0VendorRqstRead8Bytes:
    cmp  a,06h
    jnz  USBEventEP0Stall                    ; No
```

```
    ;*********************************************
    ; Read 8 Bytes Event
    ;*********************************************
    mov  a,[USBEndP0FIFO_2]   ;Port output value
    mov [gbPort0],a
    mov  a,[USBEndP1FIFO_1]
    mov [USBEndP0FIFO_1],a
    mov  a,[USBEndP1FIFO_2]
    mov [USBEndP0FIFO_2],a
    mov  a,[USBEndP1FIFO_3]
    mov [USBEndP0FIFO_3],a
    mov  a,[USBEndP1FIFO_4]
    mov [USBEndP0FIFO_4],a
    mov  a,[USBEndP1FIFO_5]
    mov [USBEndP0FIFO_5],a
    mov  a,[USBEndP1FIFO_6]
    mov [USBEndP0FIFO_6],a
    mov  a,[USBEndP1FIFO_7]
    mov [USBEndP0FIFO_7],a
    jmp  USBEventEP0VendorRqstFinish
```

Listing 6.5: Übertragung von 7 Nutzdaten in USB_20j.ASM

Das fertige Programm findet sich als USB_20j.asm auf der CD zum Buch. Um die neue Funktion 6 nutzen zu können, muss der Treiber Beyond.sys von Craig Peacock geladen werden, der weiter unten in Kap. 7 genauer vorgestellt wird.

Die neue IO-Controll-Funktion 8 wurde mit dem Device Driver Fiddler erprobt. Man erkennt in *Abb. 6.8*, dass acht Datenbytes zurückgeliefert werden.

Nach wie vor können Ergebnisse auch über direkte RAM-Zugriffe gelesen werden. Das Programm AD1.FRM demonstriert den Zugriff, wobei der Anfangskanal auf Null gesetzt wurde. Die analogen Eingänge A bis D entsprechen also den digitalen Eingängen 0 bis 3. Diese Abfrage ist auch mit dem Treiber von Cypress möglich.

```
Function RdRAM(Adr) As Integer
  lIn = Adr * 256 + 22
  lInSize = 2
  lOutSize = 2
  USB_IO
  RdRAM = (lOut / 256) And 255
End Function

Private Sub Timer1_Timer()
  WrRAM &H34, 0
  Din = RdRAM(&H37)
  Label10.Caption = Str$(Din)
  ul = 4 * RdRAM(&H7A) + (RdRAM(&H7C) And 15)
```

```
  Label7.Caption = Str$(u1)
  u2 = 4 * RdRAM(&H7B) + RdRAM(&H7C) \ 16
  Label8.Caption = Str$(u2)
  u3 = 4 * RdRAM(&H7D) + (RdRAM(&H7F) And 15)
  Label9.Caption = Str$(u3)
  u4 = 4 * RdRAM(&H7E) + RdRAM(&H7F) \ 16
  Label2.Caption = Str$(u4)
End Sub
```

Listing 6.6: Auslesen der Daten über RdRAM in TLC1543.FRM

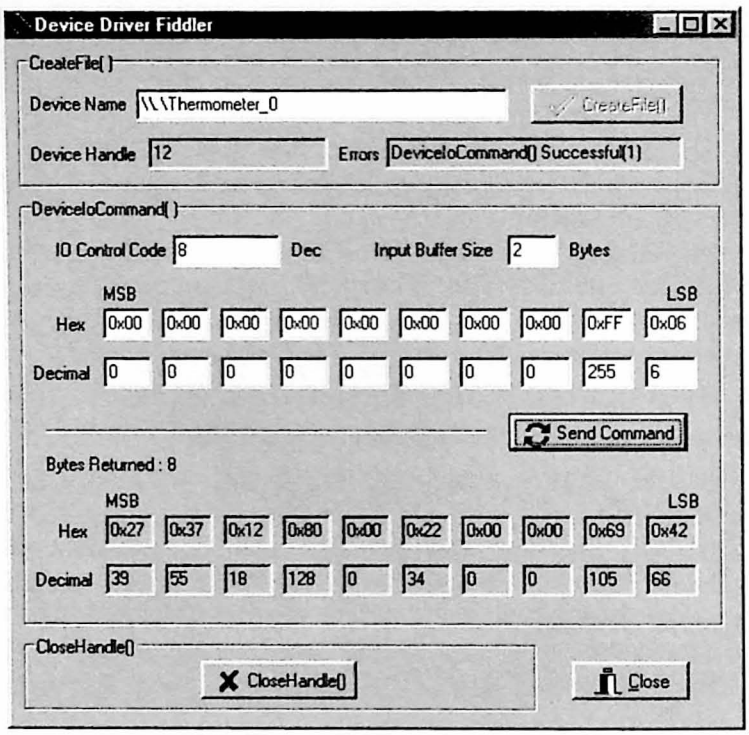


Abb. 6.8: Austausch über die interne Funktion 6

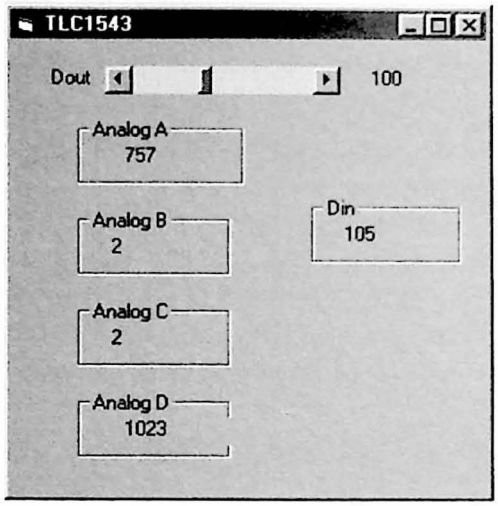


Abb. 6.9: Ausgabe der Eingangsdaten

6.5 Programmierung mit Delphi

Während Visual Basic sich für schnelle Versuche eignet, lassen sich komplexere Programme oft leichter mit Delphi schreiben. Das hier vorgestellte Programm zeigt einen ersten Zugriff auf den USB mit Delphi.

Das Programmbeispiel nach Listing 6.7 verwendet alle Ein- und Ausgänge des Interfaces. Alle Daten werden mit einem einzelnen DeviceIoCommand ausgetauscht. Die Datenstrukturen lIn mit einer Länge von vier Bytes und lOut mit einer Länge von 8 Bytes reichen für alle denkbaren Anwendungen aus.

DeviceIoCommand verwendet hier die neue Funktionsnummer 8, um einen kompletten Austausch der Daten in drei Millisekunden zu erreichen. Als erstes Byte wird die neu hinzugekommene Funktionsnummer 6 an den Controller übertragen. An der zweiten Position wird der aktuelle Wert des digitalen Ausgangsports gesendet.

Das Interface sendet insgesamt acht Bytes zurück, die in der Datenstruktur lOut erscheinen. Die Umrechnung in 10-Bit Analogwerte und in reale Spannungen entspricht dem vorangegangenen Beispiel in Visual Basic. Das Programm CompUSB verwendet bereits den weiter unten beschriebenen CompuLAB-Treiber mit dem Symbolischen Namen '\\.\CompuLABusb_0' statt dem bisher verwendeten Namen '\\.\Thermometer_0'.

```
unit CompUSB;
interface
uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, StdCtrls;
```

```
type
  TForm1 = class(TForm)
    CheckBox1: TCheckBox;
    CheckBox2: TCheckBox;
    CheckBox3: TCheckBox;
    CheckBox4: TCheckBox;
    CheckBox5: TCheckBox;
    CheckBox6: TCheckBox;
    CheckBox7: TCheckBox;
    CheckBox8: TCheckBox;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    Timer1: TTimer;
    Label9: TLabel;
    CheckBox9: TCheckBox;
    CheckBox10: TCheckBox;
    CheckBox11: TCheckBox;
    CheckBox12: TCheckBox;
    CheckBox13: TCheckBox;
    CheckBox14: TCheckBox;
    CheckBox15: TCheckBox;
    CheckBox16: TCheckBox;
    Label10: TLabel;
    Edit1: TEdit;
    Label11: TLabel;
    Edit2: TEdit;
    Label12: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  end;

type _lIn  = record
    bFunction : Byte;
    bValue1 : Byte;
    bValue2 : Byte;
    bValue3 : Byte;
end;

type _lOut  = record
    bAck : Byte;
    bValue1 : Byte;
    bValue2 : Byte;
    bValue3 : Byte;
    bValue4 : Byte;
    bValue5 : Byte;
```

```
    bValue6 : Byte;
    bValue7 : Byte;
end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
  Timer1.Interval := 100;
  Timer1.Enabled := true;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
var Wert: Word;
    DeviceHandle: THandle;
    TemplateHandle: THandle;
    nBytes: DWord;
    bresult: Boolean;
    Dout, Din, Ain: Byte;
    lIn: _lIn;
    lOut: _lOut;
    AnA, AnB: Real;
begin
  Dout := 0;
  if  CheckBox1.Checked Then Dout := Dout +1;
  if  CheckBox2.Checked Then Dout := Dout +2;
  if  CheckBox3.Checked Then Dout := Dout +4;
  if  CheckBox4.Checked Then Dout := Dout +8;
  if  CheckBox5.Checked Then Dout := Dout +16;
  if  CheckBox6.Checked Then Dout := Dout +32;
  if  CheckBox7.Checked Then Dout := Dout +64;
  if  CheckBox8.Checked Then Dout := Dout +128;

  lIn.bFunction := 6;
  lIn.bValue1 := Dout;

  DeviceHandle := CreateFile ('\\.\CompuLABusb_0',Generic_write,File_Share_write,nil,
       open_existing,0,TemplateHandle);
  bResult := DeviceIoControl(DeviceHandle,$08,@lIn,sizeof(lIn),
       @lOut,sizeof (lOut),nBytes,nil);
  CloseHandle (DeviceHandle);

  Din := lOut.bValue1;
  AnA := (4*lOut.bValue2+ (lOut.bValue4 and 15))/1023*5;
  AnB := (4*lOut.bValue3+ (lOut.bValue4 div 15))/1023*5;
  CheckBox9.Checked := ((Din And 1) >0);
  CheckBox10.Checked := ((Din And 2) >0);
```

```
  CheckBox11.Checked := ((Din And 4) >0);
  CheckBox12.Checked := ((Din And 8) >0);
  CheckBox13.Checked := ((Din And 16) >0);
  CheckBox14.Checked := ((Din And 32) >0);
  CheckBox15.Checked := ((Din And 64) >0);
  CheckBox16.Checked := ((Din And 128) >0);
  Edit1.Text := FloatToStrF(AnA,ffGeneral,3,2) + ' V';
  Edit2.Text := FloatToStrF(AnB,ffGeneral,3,2) + ' V';
end;

end.
```

Listing 6.7: Zugriff auf alle Ein- und Ausgänge

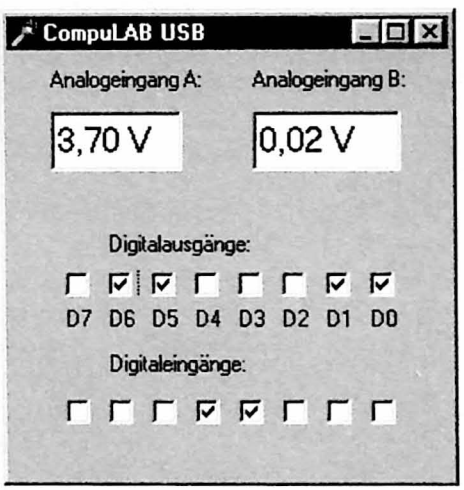


Abb. 6.10: Die Bedienoberfläche zum CompuLAB-USB

Das Programm nach *Abb. 6.10* erlaubt den vollen Zugriff auf alle Ein- und Ausgänge. Einzelne Ausgänge können durch Anklicken umgeschaltet werden. Die Zustände der digitalen Eingänge und der Analogeingänge A und B werden direkt angezeigt. Das Programm bietet eine Untermenge der Funktionen der Anwendersoftware Compact2000, die in Kap. 8 vorgestellt wird.

6.6 Elektrische Eigenschaften der Interface-Ports

Die digitalen Ausgänge des Prozessors können nicht sinnvoll direkt eingesetzt werden, da sie zu gering belastbar sind und insbesondere keinen angemessenen Strom liefern können. Die Sink-Fähigkeit kann bis auf 1.5 mA gesteigert werden, während

der Source-Strom durch die relativ hochohmigen Ausgänge auf wenige Mikroampere beschränkt ist. Es ist also in jedem Fall ein Ausgangstreiber erforderlich.

Für die Ausgänge besteht die zusätzliche Forderung nach einer optischen Zustandsanzeige durch acht LEDs. Die LEDs direkt an die Ausgänge der Porttreiber anzuschließen, hätte den Nachteil, dass die Ausgangsspannung geringfügig einbrechen würde. Für den experimentiellen Einsatz ist es aber wünschenswert, wenn die Ausgangszustänge exakt durch 0 V und +5 V repräsentiert werden.

Die LEDs sollten also direkt vom Mikrocontroller getrieben werden, damit der volle Ausgangspegel vom Ausgangstreiber restauriert werden kann. Der Port 0 des Controllers kann aber LEDs nur invertiert ansteuern. Es müssen stromsparende LEDs bei ca. 1 mA eingesetzt werden. Die LEDs werden mit ihren Vorwiderständen gegen +5 V angeschlossen und invertiert angesteuert. Der Ausgangstreiber muss daher ebenfalls die Ausgangssignale invertieren. Im Interesse eines einfachen Layouts wurde ein 74HC540 gewählt. Laut Datenblatt liefert das IC in beiden Richtungen Ströme bis 35 mA. Versuche haben darüber hinaus gezeigt, dass die Ausgänge kurzzeitig kurzschlussfest sind und sogar Berührung mit Fremdspannung vertragen, wenn der Strom auf ca. 100 mA begrenzt ist.

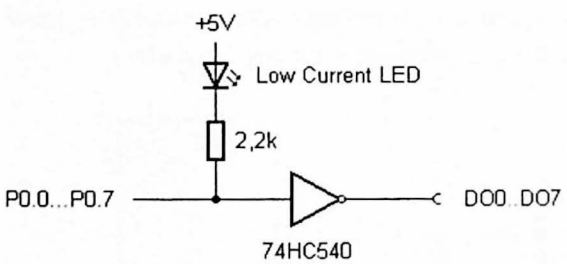


Abb. 6.11: Ausgangsbeschaltung mit einem 74HC540

In der endgültigen Version der Firmware (CLUSB02.ASM) wurde der Sink-Strom der Eingänge vergrößert, damit trotz der Belastung der Portausgänge mit den LED-Strömen genügend kleine Low-Pegel gewährleistet sind.

```
; Maximum sink current for Port 0 Pins
    iowr SysPort0ISinkPin0
    iowr SysPort0ISinkPin1
    iowr SysPort0ISinkPin2
    iowr SysPort0ISinkPin3
    iowr SysPort0ISinkPin4
    iowr SysPort0ISinkPin5
    iowr SysPort0ISinkPin6
    iowr SysPort0ISinkPin7
```

Listing 6.8: Einstellung der DA-Wandler zu Port 0

Zusätzlich müssen bei dieser Beschaltung die Port-0-Ausgaben invertiert werden. Die Inverter im 74HC540 kehren die Ausgangspegel wieder um.

USBEventEP0VendorRqstRead8Bytes:

```
    cmp  a,06h
    jnz  USBEventEP0Stall                    ; No

    ;*******************************************
    ; Read 8 Bytes Event
    ;*******************************************
    mov  a,[USBEndP0FIFO_2]   ;Port output value
    cpl  a
    mov [gbPort0],a
...
```

Listing 6.9: Invertierung der Ausgangszustände

Die Gestaltung der Eingänge richtet sich nach den Forderungen eines Überspannungsschutzes und bester Genauigkeit der analogen Messungen. In jedem Fall müssen Schutzwiderstände vorgesehen werden, die den maximalen Strom bei Überspannung und bei impulsartiger Entladung begrenzen. Die Art der Eingangsbeschaltung hängt eng mit der verwendeten Referenzspannung zusammen.

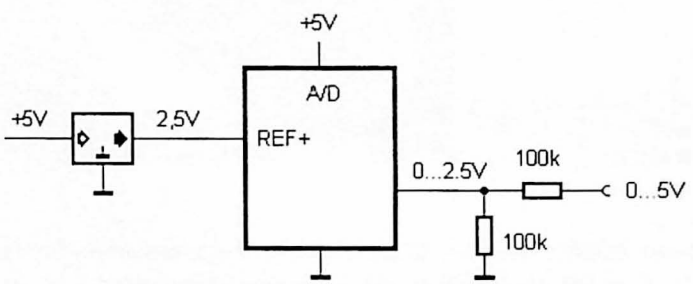


Abb. 6.12: Verwendung einer Referenz von 2,5 V

Abb. 6.12 zeigt eine mögliche Beschaltung bei einer Referenzspannung von 2,5 V. Dieses Verfahren hat den Vorteil, dass sich die Referenzspannung leicht aus der Versorgungsspannung des USB gewinnen lässt. Auch bei einem maximalen Abfall der Versorgungsspannung auf 4,2 V kann eine genaue Referenz garantiert werden. Die Eingangs-Spannungsteiler im Verhältnis 2 zu 1 heben den Messbereich wieder auf 5 V an. Die beiden Widerstände erfüllen zugleich die Forderung nach einer Strombegrenzung bei Überspannung und nach eine Leerlaufspannung von Null. Eine identische Eingangsbeschaltung kann für alle digitalen und analogen Eingänge verwendet werden.

Ein Nachteil dieser Schaltung ist jedoch eine Verschlechterung der Eingangsgenauigkeit durch Toleranzen der Widerstände. Selbst genaue Metallfilmwiderstände mit einer Toleranz von 1 % könnten im ungünstigsten Fall einen Messfehler von 2 % verursachen. Dem steht eine relative Auflösung des Wandlers von ca. 0,1% des Endbereichs gegenüber. Zwar wird eine absolute Genauigkeit in dieser Größenordnung wegen der Toleranz der Referenz nicht erreichbar sein. Wohl aber kann man eine entsprechend gute Übereinstimmung zwischen den Kanälen erreichen. Diese würde durch den Einsatz des Spannungsteilers verschlechtert. Für die praktische Realisierung des Projekts muss auch bedacht werden, dass bei insgesamt 12 Kanälen 24 Widerstände benötigt werden. Man wird daher Widerstands-Arrays einsetzen, die jedoch meist eine Toleranz von 2 % aufweisen.

Das Problem der Widerstandstoleranzen besteht nicht mehr, wenn man auf eine Spannungsteilung verzichtet. Die Eingangsschaltung nach Abb. 6.13 besteht aus einem Längswiderstand zur Strombegrenzung bei Überspannung und aus einem zusätzlichen Widerstand parallel zum Eingangsanschluss, der nur einen definierten Eingangswiderstand bildet. Diese Beschaltung verschlechtert die Genauigkeit des Wandlers nicht. Sie erfordert aber den Einsatz einer Referenzpannung von 5 V, was wegen der möglichen Spannungsabfälle auf den Versorgungsleitungen zu größeren Ungenauigkeiten führen kann.

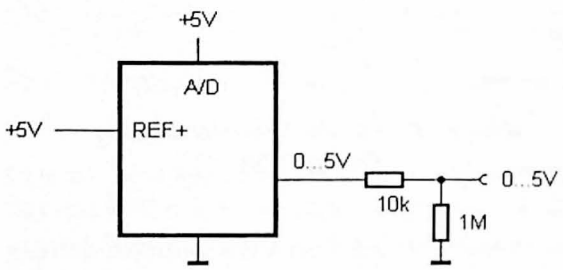


Abb. 6.13: Eingangsbeschaltung ohne Spannungsteiler

Alle digitalen und analogen Eingänge erhalten die gleiche Beschaltung. Der Eingangswiderstand ist einheitlich 1 MΩ. Der Längswiderstand von 10 kΩ hat nur eine Schutzfunktion. Bei Überspannung begrenzt er den Strom durch die internen Schutzdioden der Eingänge des AD-Wandlers. Überspannungen von +/-30V werden schadlos überstanden.

6.7 Stromversorgung

Die Frage nach der Spannungsversorgung besteht auch unabhängig von der Problematik der Referenzspannung. Das Interface soll auch eine Ausgangsbuchse für +5 V

besitzen, die bis maximal 50 mA belastbar sein sollte. Dies ermöglicht einfache Versuche ohne zusätzliche Stromversorgung und ist besonders in der Ausbildung wegen der größeren Fehlersicherheit wichtig. Zusätzlich besitzt der RS232-Vorgänger des Geräts eine auf einen Sammelstecker durchgeschleifte 12-V-Leitung zur Versorgung größerer Modelle mit einem Strombedarf von bis zu 500 mA.

Sinnvoll ist es, wenn man am USB sowohl bus-powered als auch self-powered arbeiten kann. Der Anwender wird also nach Möglichkeit ohne ein zusätzliches Netzteil arbeiten und die Versorgungsspannung vom USB verwenden. Erst wenn mehr Leistung benötigt wird, kommt ein zusätzliches Steckernetzteil zum Einsatz. Dieses kann dann die höhere Versorgungsspannung für zusätzliche Geräte liefern.

Im Normalfall kann die 5-V-Versorgung direkt vom USB aus erfolgen. Da die Spannung für den Anwender herausgeführt wird, muss unbedingt eine Strombegrenzung vorgesehen werden. Am besten eignet sich dazu eine Polyswitch-Sicherung mit einem Nennwert von 100 mA. Wenn im Normalfall nur sehr wenig Strom entnommen werden soll, reicht auch ein einfacher Widerstand von 50 Ω. Will man kleinere Spannungsabfälle erreichen, kann man den Widerstand bis auf 10 Ω verkleinern. Im Falle eines Kurzschlusses würde dann ein maximaler Strom von 500 mA fließen, was die Zerstörung sicher verhindert.

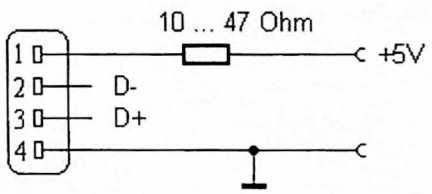


Abb. 6.14: Einfache Stromversorgung über den USB

Die einfache Schaltung nach *Abb. 6.14* verwendet die USB-Versorgungsspannung direkt. Messungen zeigen in den meisten Fällen eine Spannung von etwa 5,0 V, die intern vom PC-Netzteil erzeugt wird. Solange keine anderen großen Verbraucher am USB arbeiten, hat man also eine brauchbare Referenz. Allerdings lassen die USB-Spezifikationen Spannungsabfälle bis 4,2 V zu. Die Genauigkeit würde also stark nachlassen. In einigen Fällen spielt das keine so große Rolle. Wenn man den AD-Wandler z.B. zusammen mit Widerstandssensoren in Spannungsteilershaltung betreibt, hebt sich der Fehler auf, da dieselbe Referenz verwendet wird. Beim Messen einer unabhängigen Spannung würde das scheinbare Messergebnis jedoch mit sinkender Versorgungsspannung ansteigen.

Versuche mit einer Betriebsspannung von nur 4,2 V zeigten, dass der Mikrocontroller weniger steile Flanken beim Schalten der Ports erzeugte. Dies führte zunächst zu Störungen beim Betrieb des AD-Wandlers. Es wurde nötig, durch zusätz-

liche NOP-Befehle längere Impulse zu verwenden. Damit war eine Funktionssicherheit über den ganzen Bereich der möglichen Betriebsspannungen erreicht.

Besser als ein Widerstand arbeitet eine Polyswitch-Sicherung, weil sie im kalten Zustand einen sehr geringen Widerstand von unter 5 Ohm aufweist. Trotzdem kommt es noch zu Spannungsabfällen, die die Genauigkeit beeinträchtigen. Für das CompuLAB-USB wurde zusätzlich eine Versorgungsbuchse für 12 V und ein Spannungsregler vorgesehen (vgl. *Abb. 6.15*). Man kann bei geringen Anforderungen an die Genauigkeit ohne externes Netzteil arbeiten. Für größere Lastströme oder bessere Genauigkeit wählt man dagegen die externe Versorgung.

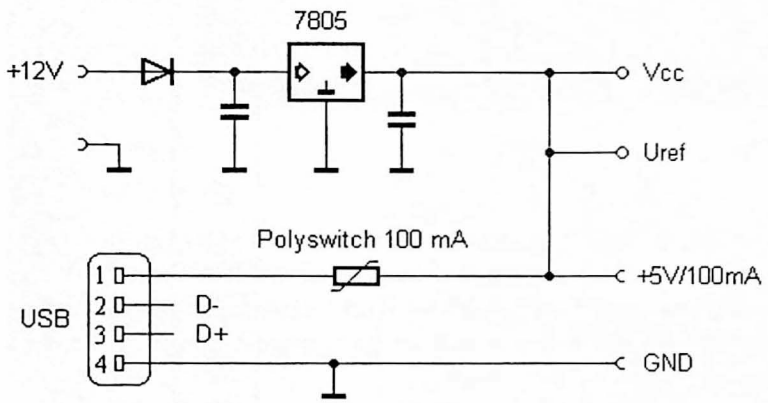


Abb. 6.15: Wahlweise Versorgung über den USB oder eine externe Spannungsquelle

Es wird keine spezielle Umschaltung vorgesehen, so dass bei kleiner Spannung am Bus sogar Strom an den USB geliefert wird. Dieser Fall tritt jedoch nur ein, wenn weitere Verbraucher am Bus größere Spannungsabfälle verursachen. Die Polyswitch-Sicherung begrenzt im Extremfall den zurückfließenden Strom.

6.8 Das CompuLAB-USB

Die Entwicklung des CompuLAB-USB wurde hier in kleinen Schritten beschrieben. Der vorgestellte Stand der Entwicklung reicht bis zur Version 0.20. *Abb. 6.16* zeigt den fertigen Prototyp. Das Gehäusedesign lehnt sich weitgehend an das schon bekannte CompuLAB für die RS232-Schnittstelle an. Die wesentliche sichtbare Änderung betrifft den USB-Anschluss und die ihm zugeordnete grüne LED zur Anzeige des enumerierten Zustands.

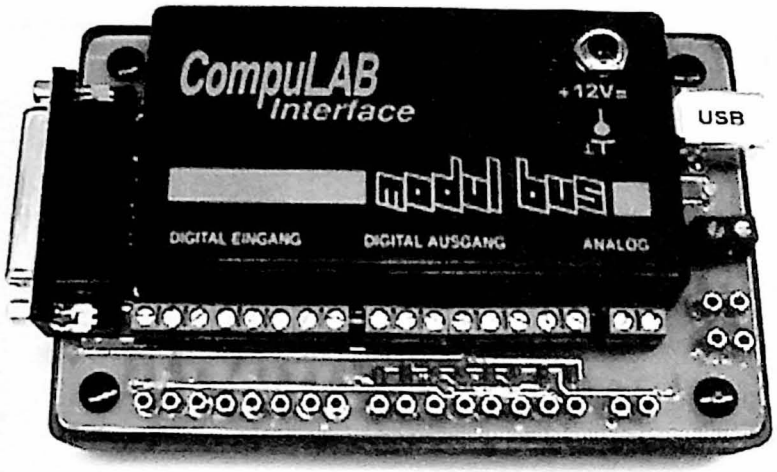


Abb. 6.16: Ein Prototyp des CompuLAB-USB

Abb. 6.18 zeigt die komplette Schaltung des Geräts. Die einzelnen Baugruppen entsprechen im Wesentlichen den bisher gezeigten Detailschaltbildern. Wie das Vorgängermodell besitzt das neue CompuLAB-USB acht digitale Eingänge, acht digitale Ausgänge mit Zustands-LEDs und zwei Analogeingänge A und B. Außerdem ist die Versorgungsspannung von 5 V zugänglich.

Aus Gründen der Kompatibilität wurde der letzte Kanal der AD-Wandlers (A10 = Eingang C) nicht an eine Buchse gelegt, sondern nur am Sammelstecker des Geräts herausgeführt. Dieser Eingang nimmt eine Sonderstellung ein, weil der Eingangswiderstand von 1 MΩ mit Absicht weggelassen wurde. Man erhält so einen Eingang mit extrem hohem Innenwiderstand. Leerlaufmessungen zeigen daher im allgemeinen keine Spannung von Null. Die Standardsoftware soll diesen Eingang daher nicht anzeigen. Er steht vielmehr für besondere Anwendungen bereit, bei denen es auf einen großen Innenwiderstand ankommt.

Eine weitere Besonderheit betrifft die Versorgung der Zustands-LEDs. Sie stellen je nach Schaltzustand des Ausgangs eine wechselnde Belastung dar. Diese sollte sich möglichst wenig auf die Betriebsspannung auswirken, auch wenn man ohne externes Netzteil arbeitet. Die LEDs werden daher direkt vom 5-V-Anschluss am USB-Stecker versorgt. Wechselnde Ströme führen dadurch zu kleinen wechselnden Spannungsabfällen am geringen Innenwiderstand der Spannungsversorgung des USB von bis herunter zu etwa einem Ohm.

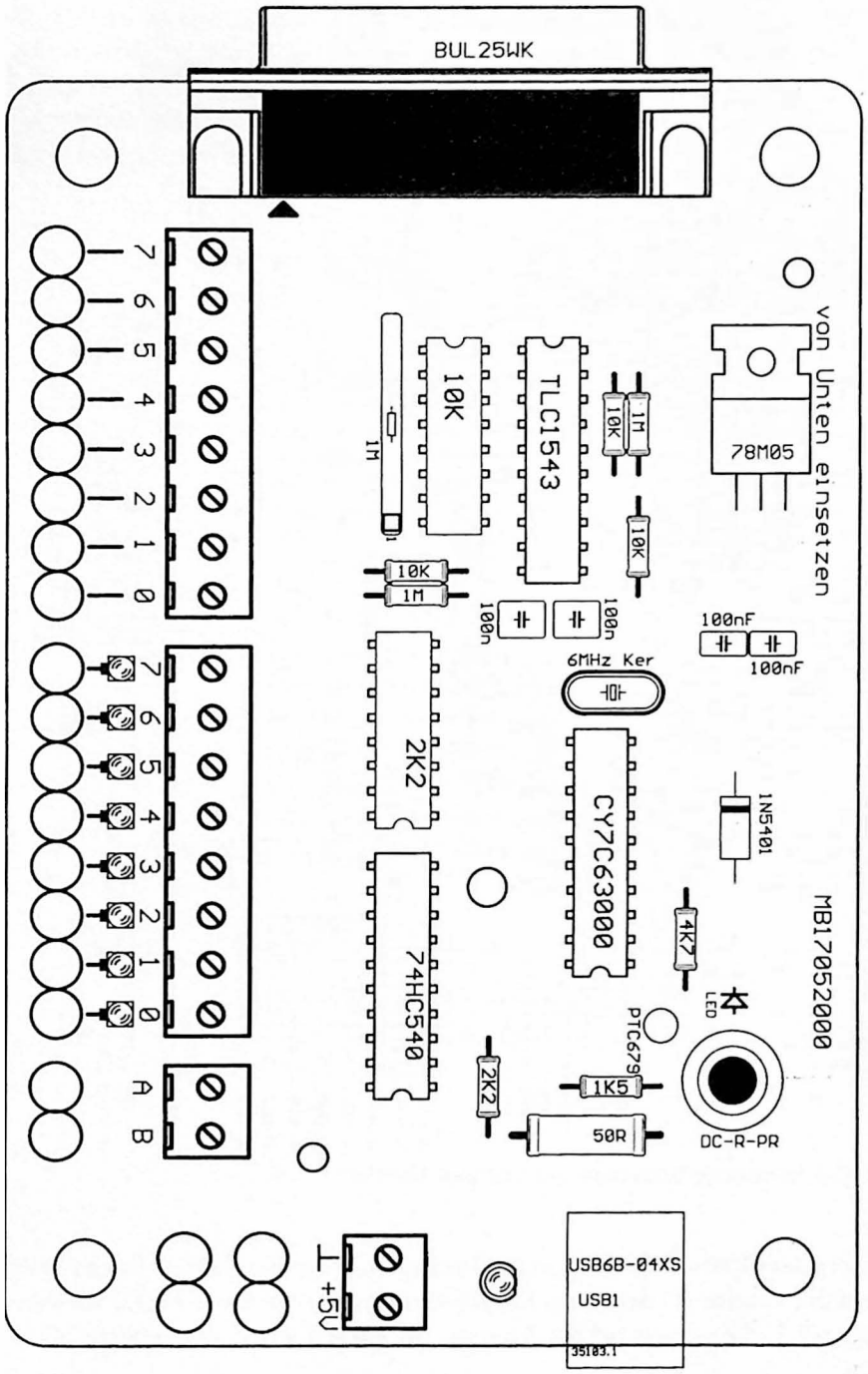


Abb. 6.17: Das Platinenlayout

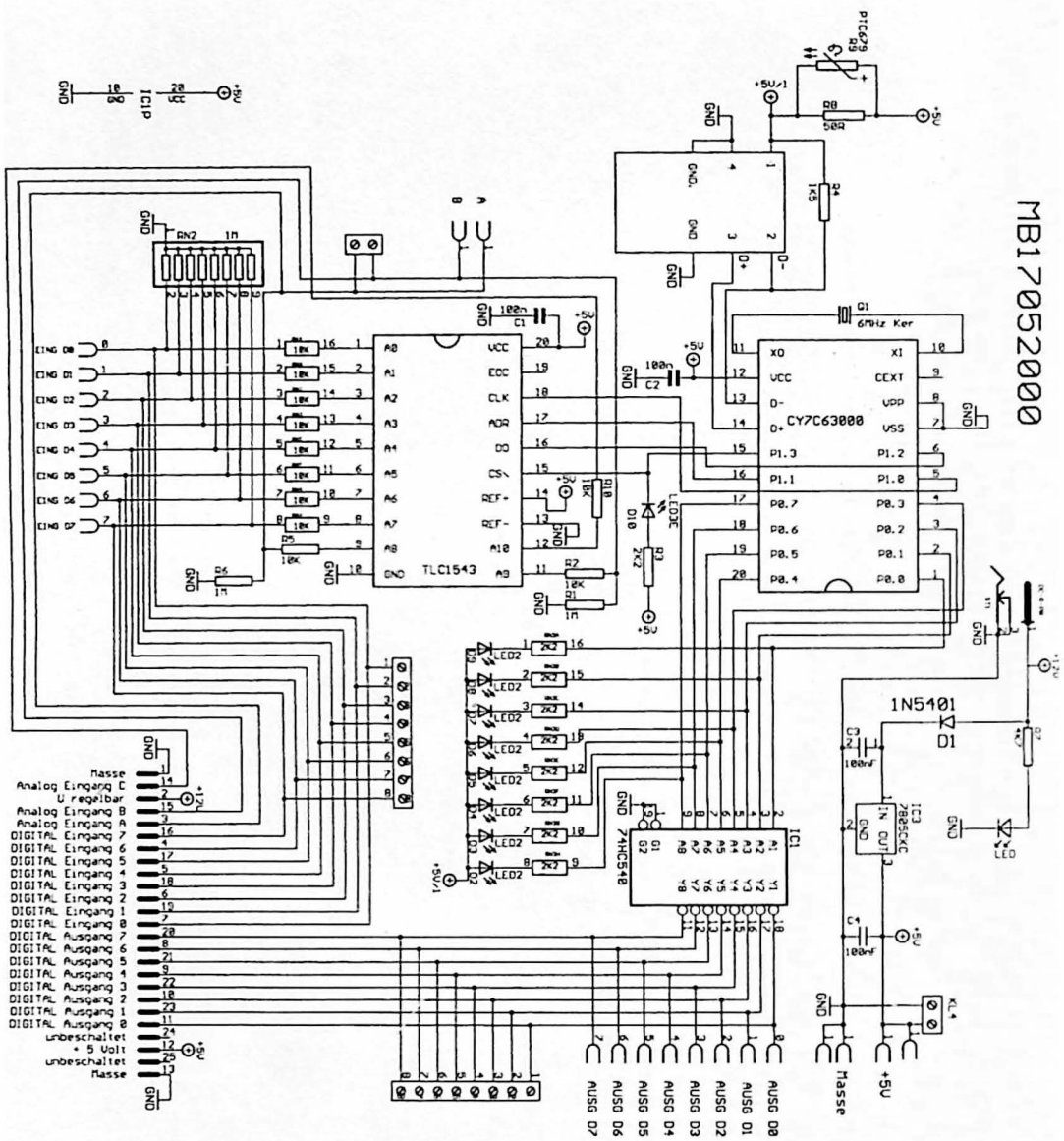


Abb. 6.18: Das komplette Schaltbild des CompuLAB-USB

Die Firmware des Geräts findet man in der Version 0.20 auf der CD des Buchs. Hier wird bereits die Vendor-ID der Firma Modul-Bus und die Device-ID 0002 verwendet (vgl. Kap. 7.2). Dies muss bei der Auswahl des passenden Treibers berücksichtigt werden.

Man kann davon ausgehen, dass es noch einige weitere Versionen der Firmware geben wird, bis alle „Kinderkrankheiten" des Geräts beseitigt sind. Die hier vorgestellte Entwicklung stellt aber einen Stand dar, der eigene Experimente ermöglicht. Prinzipiell kann jeder erfahrene Hobby-Elektroniker das Gerät aufbauen und mit der vorhandenen Software betreiben.

7 Treiber

Bei den bisherigen Versuchen wurde zunächst der Thermometer-Treiber von Cypress verwendet. Auch die erweiterten Anwendungen liefen als „Thermometer". Eine komplett eigene Lösung benötigt auch einen eigenen Treiber. Speziell für das neu entwickelte CompuLAB-USB wurde ein neuer Treiber benötigt. Diese Aufgabe erfordert einige Vorkenntnisse in C-Programmierung. Als Werkzeuge benötigt man Visual C++ und das Driver Development Kit für Windows98 (DDK98) von Microsoft.

Bei der Entwicklung eines Treibers konnte ich auf Vorarbeiten und Hilfen von Craig Peacock und Hans-Joachim Berndt zurückgreifen. Beiden sei an dieser Stelle herzlich gedankt.

7.1 Umbau eines Beispiels-Treibers

In der DDK89 findet man Beispiele zu USB-Treibern. Am einfachsten ist es, von einem bestehenden Beispiel auszugehen und nur die entscheidenden Stellen abzuändern. Craig Peacock (www.beyondlogic.org) schrieb einen eigenen Treiber durch Umbau des ISOUSB-Beispiels zum Intel-Microcontroller 82930 aus der DDK98. Dieser Treiber ist auf der CD enthalten. Er ist noch speziell auf das USB-Thermometer zugeschnitten, erlaubt aber bereits Vendor-Commands mit einer Übertragung von 8 Bits. Der Treiber Beyond.sys lässt sich mit der Datei CypressUSB.INF laden und ersetzt den ursprünglichen Treiber von Cypress.

Die erste veränderte Datei ist das Hauptmodul ISOUSB.C. Hier wird unter anderem ein symbolischer Dateiname für den Treiber aufgebaut, der später mit CreateFile verwendet wird. Wegen der gewünschten Kompatibilität zum Treiber von Cypress wurde der Name „Thermometer_0" verwendet. Hier kann leicht ein anderer Name eingesetzt werden.

```
/*++
Copyright (c) 1997-1998  Microsoft Corporation
Module Name:
    IsoUsb.c
```

```
{
    NTSTATUS ntStatus;
    UNICODE_STRING deviceLinkUnicodeString;
    PDEVICE_EXTENSION deviceExtension;
    USHORT i;

    UNICODE_STRING pdoUniCodeName;
    WCHAR      pdoName[] = L"\\Device\\Thermometer_0";

    UNICODE_STRING DeviceLinkUniCodeString;
    WCHAR          DeviceLinkName[] =
       L"\\DosDevices\\Thermometer_0";

    RtlInitUnicodeString (&pdoUniCodeName, pdoName); // Creates
Unicode Name

    ntStatus = IsoUsb_SymbolicLink( PhysicalDeviceObject,
 &deviceLinkUnicodeString );
    ntStatus = IoCreateDevice (DriverObject, // Driver Object
                sizeof (DEVICE_EXTENSION),   // Size
                &pdoUniCodeName,             // Name
                FILE_DEVICE_UNKNOWN,
                0,          // No Special Characteristics
                FALSE,
                DeviceObject);

    if (NT_SUCCESS(ntStatus))
        {
             deviceExtension = (PDEVICE_EXTENSION)
                ((*DeviceObject)->DeviceExtension);
        }

    if (!NT_SUCCESS(ntStatus))
        {
             return ntStatus;
        }

RtlInitUnicodeString (&DeviceLinkUniCodeString,
           &DeviceLinkName);
ntStatus = IoCreateSymbolicLink(&DeviceLinkUniCodeString,
             &pdoUniCodeName);

...
```

Listing 7.1: Erzeugen eines symbolischen Dateinamens für den Treiber

Der symbolische Treibername muss auch noch in die Datei IsoPnp.c eingetragen werden (vgl. Listing 7.2).

```
/*++

Copyright (c) 1997-1998  Microsoft Corporation

Module Name:

    IsoPnp.c

Abstract:

    Isochronous USB device driver for Intel 82930 USB test board
    Plug and Play module

...

--*/
{

    PIO_STACK_LOCATION irpStack;
    PDEVICE_EXTENSION deviceExtension;
    NTSTATUS ntStatus = STATUS_SUCCESS;
    NTSTATUS waitStatus;
    PDEVICE_OBJECT stackDeviceObject;
    KEVENT startDeviceEvent;
    UNICODE_STRING DeviceLinkUniCodeString;
    WCHAR          DeviceLinkName[] =
L"\\DosDevices\\Thermometer_0";


    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;

...
```

Listing 7.2: Änderung in ISOPNP.C

Allgemein werden in den beschriebenen Anwendungen nur DeviceIoControl-Zugriffe über Endpoint 0 durchgeführt. Daher betreffen die entscheidenden Änderungen in der Funktionalität des Treibers nur die Datei IOCTLISO.C. Listing 7.3 zeigt die entscheidenden Anpassungen von Craig Peacock.

```
/*++
Module Name:
   ioctliso.c
--*/

NTSTATUS
```

```
IsoUsb_ProcessIOCTL(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
/*++

Routine Description:
    Dispatch table handler for IRP_MJ_DEVICE_CONTROL;
    Handle DeviceIoControl() calls  from User mode

Return Value:

    NT status code

--*/
{
    PIO_STACK_LOCATION irpStack;
    PVOID ioBuffer;
    ULONG inputBufferLength;
    ULONG outputBufferLength;
    PDEVICE_EXTENSION deviceExtension;
    ULONG ioControlCode;
    NTSTATUS ntStatus;
    ULONG length;
    PUCHAR pch;
    PURB urb;
    USHORT temp;
    USHORT temp2;
    PUSB_CONFIGURATION_DESCRIPTOR configurationDescriptor;

    IsoUsb_IncrementIoCount(DeviceObject);

    deviceExtension = DeviceObject->DeviceExtension;

    if ( !IsoUsb_CanAcceptIoRequests( DeviceObject ) )
     {
        ntStatus = STATUS_DELETE_PENDING;
        Irp->IoStatus.Status = ntStatus;
        Irp->IoStatus.Information = 0;

        IoCompleteRequest( Irp, IO_NO_INCREMENT );

        IsoUsb_DecrementIoCount(DeviceObject);
        return ntStatus;
     }
    irpStack = IoGetCurrentIrpStackLocation (Irp);

    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;
```

```
    ioBuffer           = Irp->AssociatedIrp.SystemBuffer;
    inputBufferLength  =
      irpStack->Parameters.DeviceIoControl.InputBufferLength;
    outputBufferLength =
      irpStack->Parameters.DeviceIoControl.OutputBufferLength;
    ioControlCode =
      irpStack->Parameters.DeviceIoControl.IoControlCode;

    switch (ioControlCode)
    {
     case 4: //Implemented for Compatability with Cypress
Thermometer Driver.
   pch = (PUCHAR) ioBuffer;
   switch(pch[0])
        {
     case 0x0E: // Set LED Brightness
           temp = pch[1];      // Store Brightness
                length = VendorCommand(DeviceObject,
                0x03, // WriteRAM
                0x2C, // gbLEDBrightness
                (USHORT)pch[1],
                ioBuffer);
                     length = VendorCommand(DeviceObject,
                0x03, // WriteRAM
                0x2B, // gbLEDBrightnessUpdate
                0x01, // TRUE
                ioBuffer);
                pch[0] = 0;         //Status
           pch[1] = temp;      //Brightness
           Irp->IoStatus.Information = 2;
                     Irp->IoStatus.Status = STATUS_SUCCESS;
           ntStatus = STATUS_SUCCESS;
           break;
     case 0x0B: // Read Thermometer
           length = VendorCommand(DeviceObject,
                0x02, // ReadRAM
                0x33, // gbThermTempRead
                0,
                ioBuffer);
           temp = pch[1]; // Store Temperature
                length = VendorCommand(DeviceObject,
                0x02, // ReadRAM
                0x34, // gbThermTempRead2
                0,
                ioBuffer);
           temp2 = pch[1]; // Store Sign
                length = VendorCommand(DeviceObject,
                0x02, // ReadRAM
                0x7A, // gbButtonPushed
```

```
            0,
            ioBuffer);
        pch[3] = pch[1]; //Move Button
        pch[0] = 0;      //Status
        pch[1] = temp;   //Temperature
        pch[2] = temp2;  //Sign
        Irp->IoStatus.Information = 4;
                Irp->IoStatus.Status = STATUS_SUCCESS;

        ntStatus = STATUS_SUCCESS;
        break;

case 0x14: // Read Port
                length = VendorCommand(DeviceObject,
            0x04,   // ReadPort
            pch[1], // Address
            0,
            ioBuffer);
        pch[0] = 0;      //Status
        Irp->IoStatus.Information = 2;
                Irp->IoStatus.Status = STATUS_SUCCESS;

        ntStatus = STATUS_SUCCESS;
        break;

case 0x15: // Write Port
                length = VendorCommand(DeviceObject,
            0x05,   // WritePort
            pch[1], // Address
            ioBuffer);
        Irp->IoStatus.Information = 1;
                Irp->IoStatus.Status = STATUS_SUCCESS;
        ntStatus = STATUS_SUCCESS;
        break;

case 0x16: // Read RAM
        length = VendorCommand(DeviceObject,
            0x02,   // ReadRAM
            pch[1], // Address
            0,
            ioBuffer);
        pch[0] = 0;      //Status
        Irp->IoStatus.Information = 2;
                Irp->IoStatus.Status = STATUS_SUCCESS;
        ntStatus = STATUS_SUCCESS;
        break;

case 0x17: // Write RAM
                length = VendorCommand(DeviceObject,
```

```
            0x03,   // WriteRAM
            pch[1], // Address
            pch[2], // Value
            ioBuffer);
        Irp->IoStatus.Information = 1;
                Irp->IoStatus.Status = STATUS_SUCCESS;
        ntStatus = STATUS_SUCCESS;
        break;

   case 0x18: // Read ROM
                length = VendorCommand(DeviceObject,
            0x01,   // ReadROM
            pch[1], // Address
            0,
            ioBuffer);
        pch[0] = 0;      //Status
        Irp->IoStatus.Information = 2;
                Irp->IoStatus.Status = STATUS_SUCCESS;

        ntStatus = STATUS_SUCCESS;
        break;

    default  :
          break;
    ntStatus = STATUS_SUCCESS;
        }
       break;

    case 8: //Reads a Vendor Command

  pch = (PUCHAR) ioBuffer;
  length = VendorCommand(DeviceObject, (UCHAR)pch[0],
(USHORT)pch[1], (USHORT)pch[2], ioBuffer);
       Irp->IoStatus.Information = length;
       Irp->IoStatus.Status = STATUS_SUCCESS;
       ntStatus = STATUS_SUCCESS;
  break;

    case 12: // GetDescriptor(s)
       pch = (PUCHAR) ioBuffer;
       length = GetDescriptor(DeviceObject, (UCHAR)pch[0],
        (UCHAR)pch[1], (USHORT)pch[2], ioBuffer);
  Irp->IoStatus.Information = length;
       Irp->IoStatus.Status = STATUS_SUCCESS;
       ntStatus = STATUS_SUCCESS;
  break;

    case 16: // Get Status
       pch = (PUCHAR) ioBuffer;
```

```
   length = GetStatus(DeviceObject, (USHORT)pch[0],
(USHORT)pch[1], ioBuffer);
   Irp->IoStatus.Information = length;
        Irp->IoStatus.Status = STATUS_SUCCESS;
        ntStatus = STATUS_SUCCESS;
   break;
....
....
        default:
        ntStatus = STATUS_INVALID_PARAMETER;
        Irp->IoStatus.Status = STATUS_INVALID_PARAMETER;
    }

    IoCompleteRequest (Irp,
                       IO_NO_INCREMENT
                       );

    IsoUsb_DecrementIoCount(DeviceObject);

    return ntStatus;

}
...
```

Listing 7.3: Die Bearbeitung von DeviceIoControl-Requests

Unter DeviceIoControlCode=4 werden alle Funktionen des ursprünglichen Treibers von Cypress nachgebildet. Man kann also weiterhin mit der ursprünglichen Firmware die Temperatur auslesen, die LED-Helligkeit verstellen und Schreib- und Lesezugriffe auf das RAM und die Ports ausführen. Die Wahl des Control-Code 4 entspricht nicht den empfohlenen Konventionen (vgl. Kap 2.9), die Funktionsnummern unter 800h für Microsoft reservieren. Sie war aber hier aus Gründen der Kompatibilität zur Thermometer-Applikation erforderlich.

Zusätzlich hat Craig Peacock mit DeviceIoControlCode=8...28 weitere Funktionen eingebaut, mit denen neue Zugriffe möglich werden. Dazu gehört z.B. die Abfrage von Deskriptoren, was bei der Entwicklung hilfreich sein kann. Besonders wichtig ist aber hier die Treiber-Funktion mit DeviceIoControlCode=8 für allgemeine und beliebige Vendor-Requests. Hier werden alle acht übertragenen Bytes unverändert zurückgegeben, so dass man die maximal mögliche Datenmenge in einem Zugriff übertragen kann. Diese Möglichkeit wird im CompuLAB-USB verwendet, um digitale und analoge Eingänge zusammen abzufragen.

Der vorläufig endgültige CompuLAB-USB-Treiber wurde auf der Basis des veränderten ISOUSB-Treibers nach dem Beispiel von Craig Peacock realisiert. Die entschei-

dende Änderung betraf nur ISOUSB.C und ISOPNP.C. Hier wurde der veränderte Treiber-Name „CompuLABusb-0" eingefügt. Der kompilierte Treiber ISOUSB.SYS wurde dann in COMPULAB.SYS umbenannt. Damit er in dieser Form auch geladen werden kann, benötigt man die Date CompuLAB.INF. Beide Dateien sind mit den entscheidenden Quelltexten auf der CD vorhanden.

Der neue Treiber wurde erfolgreich unter Windows98 getestet. Bisher liegen noch keine Erfahrungen mit Windows 2000 vor. Auch besteht bisher noch nicht die Möglichkeit, mehr als ein Gerät gleichzeitig am Bus zu betreiben. Auch hier wird die Entwicklung noch weitergehen.

7.2 Anpassung der Firmware

Ein firmeneigener Treiber kann nur geladen werden, wenn das entwickelte USB-Gerät über eine entsprechende INF-Datei mit dem Treiber verknüpft wird. Die Zuordnung erfolgt über die Vendor-ID und die Product-ID des Geräts. Jede Firma kann eine VID bei der USB-Organisation beantragen. Die PID wird firmenintern vergeben. Für das CompuLAB-USB der Firma Modul-Bus lauten die beiden ID-Numern:

VID = $0A2C = 2604 (AK-Modul-Bus)
PID = $0002 = 2 (CompuLAB-USB)

Mit einer kleinen Änderung der Firmware werden diese IDs in den Device-Deskriptor eingetragen. Bei der Enumeration des Geräts werden die IDs von System gelesen, um den passenden Treiber zu laden.

```
USBDeviceDescription:
    db 12h          ; Length
    db 01h          ; Type (1=device)
    db 00h,01h      ; Complies to USB Spec. v1.00
    db 00h          ; Class code (0=??)
    db 00h          ; SubClass code (0=??)
    db 00h          ; Protocol (0=none)(9.6.1)
    db 08h          ; Max. packet size for port0
    db 2Ch,0Ah      ; Vendor ID: (0x0A2C=Modul-Bus)
    db 02h,00h      ; Product ID (0x02=USB CompuLAB)
    db 02h,00h      ; Device release v0.20
    db 01h          ; Manufacturer string descriptor index
    db 02h          ; Product string descriptor index
    db 00h          ; Serial number string descriptor index
    db 01h          ; Number of possible configurations
USBDeviceDescriptionEnd:
```

Listing 7.4: Der veränderte USB-Device-Deskriptor

Die veränderte Firmware befindet sich in der Datei CLUSB02.asm auf der CD. Auch hier ist die Entwicklung noch nicht abgeschlossen. Insbesondere befinden sich noch String-Deskriptoren der ursprünglichen Thermometer-Applikation im Programm. Auch gibt es Abschnitte in der Software, die nicht mehr wirklich verwendet werden. Die Firmware ist jedoch brauchbar, um ein eigenes CompuLAB-USB aufzubauen. Die Entwicklung der Firmware wie auch des Treibers wird nach Fertigstellung des Buchs fortgesetzt. Das mittelfristige Ziel ist die Verwendung auch mit Windows 2000 und der gemeinsame Betrieb mehrerer Geräte.

7.3 Die INF-Datei

Die erforderliche INF-Datei muss unter anderem den Treibernamen COMPULAB.SYS und die verwendeten IDs enthalten. Der genaue Aufbau kann stark variieren. Beispiele für INF-Dateien findet man in der DDK98. Die hier verwendete Datei ist von der Datei ISOUSB.INF abgeleitet. Informationen zum genauen Aufbau einer INF-Datei findet man auch im Internet auf der Microsoft-Seite.

Die Datei enthält einige obligatorische und verschiedene optionale Abschnitte. Der Abschnitt [Version] enthält einige Angaben zum Betriebssystem. [Manufacturer] gibt den Hersteller der Hardware an. In jedem Abschnitt kann ein neuer Name definiert werden, unter dem ein neuer Abschnitt geführt wird. Im der Datei CompuLAB.INF wird entsprechend unter [MODULBUS] der Manufacturer-Eintrag zur Vendor-und Product-ID geführt. Der zugeordnete Treiber ist COMPULAB.SYS.

```
[Version]
Signature="$CHICAGO$"
Class=USB
provider=%MB%
LayoutFile=layout.inf

[Manufacturer]
%MfgName%=MODULBUS

[MODULBUS]
%USB\VID_0A2C&PID_0002.DeviceDesc%=COMPULAB.Dev, USB\VID_0A2C&PID_0002

[PreCopySection]
HKR,,NoSetupUI,,1

[DestinationDirs]
COMPULAB.Files.Ext = 10,System32\Drivers
COMPULAB.Files.Inf = 10,INF
```

```
[COMPULAB.Dev]
CopyFiles=COMPULAB.Files.Ext, COMPULAB.Files.Inf
AddReg=COMPULAB.AddReg

[COMPULAB.Dev.NT]
CopyFiles=COMPULAB.Files.Ext, COMPULAB.Files.Inf
AddReg=COMPULAB.AddReg

[COMPULAB.Dev.NT.Services]
Addservice = COMPULAB, 0x00000002, COMPULAB.AddService

[COMPULAB.AddService]
DisplayName    = %COMPULAB.SvcDesc%
ServiceType    = 1                  ; SERVICE_KERNEL_DRIVER
StartType      = 2                  ; SERVICE_AUTO_START
ErrorControl   = 1                  ; SERVICE_ERROR_NORMAL
ServiceBinary  = %10%\System32\Drivers\COMPULAB.sys
LoadOrderGroup = Base

[COMPULAB.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,COMPULAB.sys
HKLM,"System\Currentcontrolset\Services\COMPULAB\Parameters",
   "MaximumTransferSize",0x10001,256
HKLM,"System\Currentcontrolset\Services\COMPULAB\Parameters",
   "DebugLevel",0x10001,2

[COMPULAB.Files.Ext]
COMPULAB.sys

[COMPULAB.Files.Inf]
COMPULAB.Inf

;------------------------------------------------------------;

[Strings]
MB="AK MODUL-BUS GmbH"
MfgName="MODULBUS"
USB\VID_0A2C&PID_0002.DeviceDesc="Modul-Bus GmbH, CompuLAB-USB"
COMPULAB.SvcDesc="CompuLAB-USB Driver"
```

Listing 7.5: Die Datei COMPULAB.INF

Anders als in der ursprünglichen INF-Datei von Cypress wird hier keine eigene Geräte-Klasse (Thermometer) gebildet, sondern das neue Gerät wird unter der bestehenden Klasse USB eingeordnet.

Beim ersten Verbinden des USB-CompuLAB erscheint eine PlugAndPlay-Meldung „Neues Gerät gefunden". Der Anwender wird aufgefordert, eine Diskette mit Treiber-Informationen einzulegen. Das System findet dann die INF-Datei und die zugehörige Treiberdatei COMPULAB.SYS. Im Geräte-Manager kann man das neue Gerät finden.

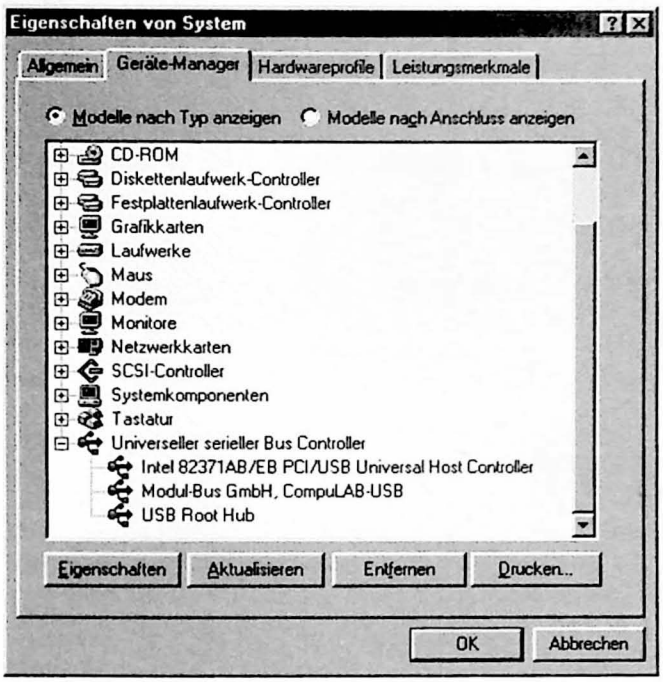


Abb. 7.1: Das enumerierte Gerät im Geräte-Manager

8 Messungen und Experimente

Das universelle Interface CompuLAB-USB eignet sich vor allem für allgemeine Versuche und Messungen in Ausbildung und Hobby. Bei der Entwicklung des Geräts wurde vor allem auf eine ausreichende Fehlertoleranz Wert gelegt. Vor allem wurde darauf geachtet, dass Fehler in einem Versuchsaufbau nicht zu einer Störung im angeschlossenen PC führen. Die Ausgänge sind daher kurzschlussfest und alle Eingänge sind weitgehend gegen Überspannungen gesichert.

Hier werden einige typische Versuche mit dem Gerät in einer kurzen Übersicht vorgestellt, um einen Eindruck von den Möglichkeiten zu vermitteln. Weitergehende Informationen dazu findet man im Internet auf der Homepage des Herstellers (www.modul-bus.de).

8.1 Das Anwenderprogramm Compact 2000

Das hier vorgestellte Programm Compact 2000 des Autors H.-J. Berndt ist eine Weiterentwicklung des bekannten Programms Do-It, das für den Vorläufer des hier beschriebenen Geräts (CompuLAB) und andere Interfaces erhältlich ist. Das Programm wurde parallel zur Hardware weiterentwickelt und liegt auf der CD zu diesem Buch in der Version 1.0 vor. Hier sollen nur die wichtigsten Eigenschaften des Programms vorgestellt werden. Compact 2000 besitzt eine automatische Hardware-Erkennung. Außer dem CompuLAB-USB werden weitere Geräte an der seriellen Schnittstelle erkannt: CompuLAB, SIOS und die ZELLE.

Im Menü Ein/Ausgänge erhält man eine Übersicht über die Spannungen an den Analogeingängen A und B und über die Zustände der digitalen Eingänge. Die digitalen Ausgänge können direkt geschaltet werden.

Messwerte der Analogeingänge werden hier als Bytes, also mit einer Auflösung von 8 Bit verarbeitet. Dies erleichtert den Umgang mit der weiter unten vorgestellten Programmierumgebung.

Das Programm bietet mit dem TY-Schreiber eine einfache Möglichkeit, Messwerte direkt grafisch aufzuzeichnen. Der Messzeitraum kann zwischen einer Sekunde und 24 Stunden gewählt werden. Man kann wahlweise einen oder zwei Kanäle aufzeichnen. Die Messung erfasst im Hintergrund immer beide Kanäle, so dass auch nachträglich in der Darstellung die Kanäle einzeln oder gemeinsam gezeigt werden können.

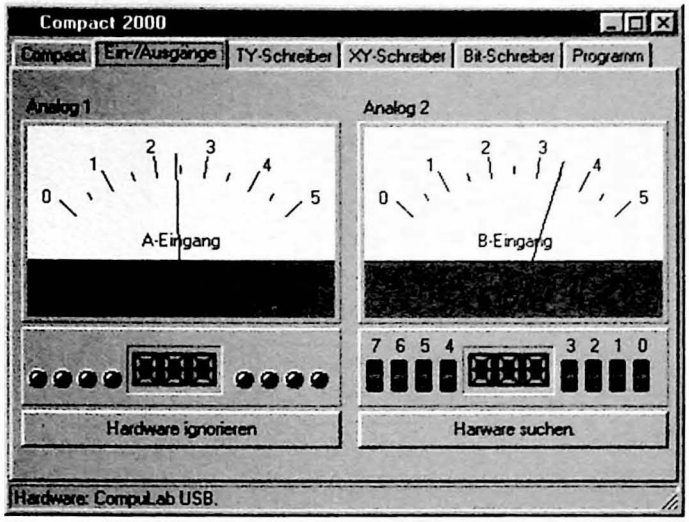


Abb. 8.1: Direktanzeige von Messwerten

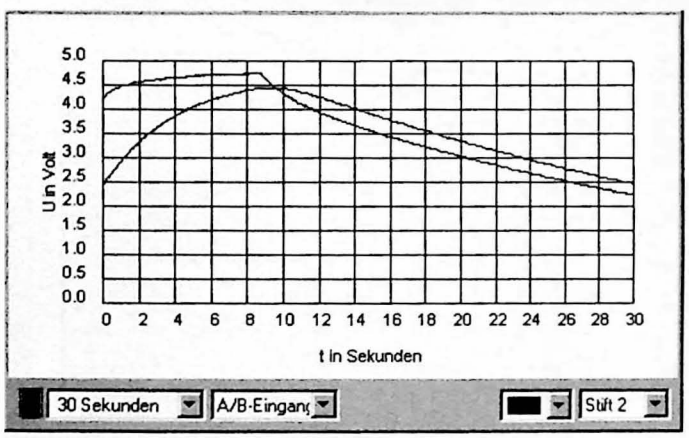


Abb. 8.2: Der TY-Schreiber

Die gewonnenen Daten können sehr einfach in andere Programme übertragen werden. Klickt man mit der rechten Maustaste in das Diagramm, dann erhält man die Auswahl „Kopieren" zur Übertragung der Grafik in die Zwischenablage oder „Kopieren als Text" zum Kopieren der Daten in Tabellenform. In der Tabellenform erhält man Messwerte in Volt und die zugehörigen genauen Zeitpunkte in Sekunden.

Tabelle 8.1:Messdaten in Tabellenform

0	4,19	2,43
0,063	4,23	2,47
0,11	4,25	2,49
0,173	4,27	2,52
0,22	4,29	2,54
0,283	4,33	2,58

Die gewonnenen Daten lassen sich problemlos in eine Excel-Tabelle einfügen und dort weiterverarbeiten. *Abb. 8.3* zeigt eine typische Darstellung der Messung in Excel.

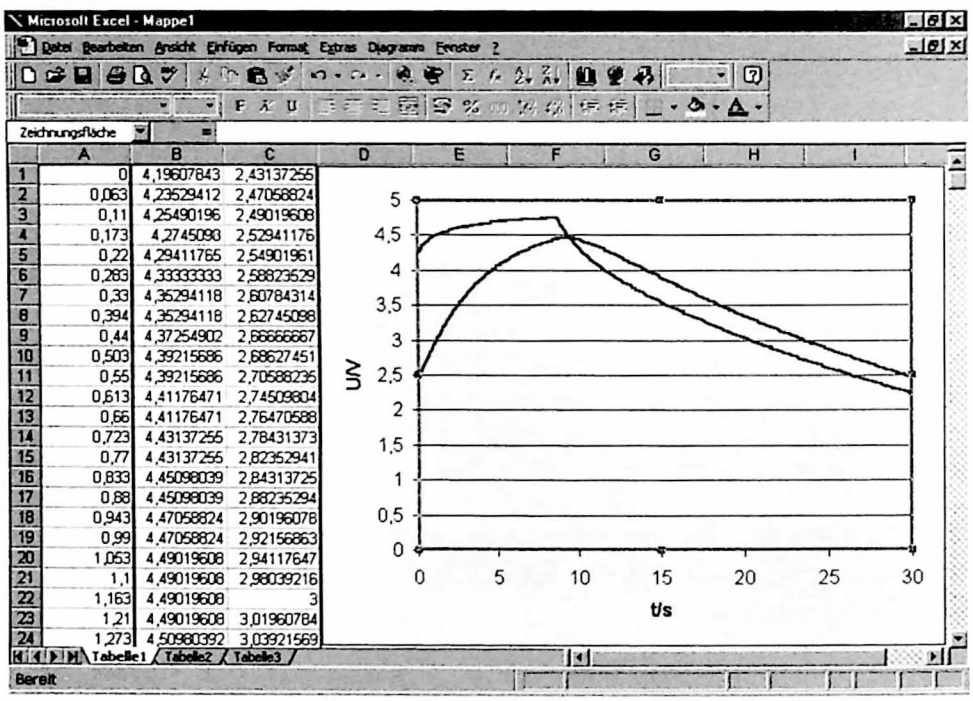


Abb. 8.3: Auswertung von Messdaten in Excel

Messdaten beider Kanäle können zur Vereinfachung verschiedener Messverfahren im XY-Schreiber gegeneinander aufgetragen werden. Man verwendet dazu einen Stift, der ein- und ausgeschaltet werden kann. Die gewonnenen Daten können als Text in andere Programme exportiert werden.

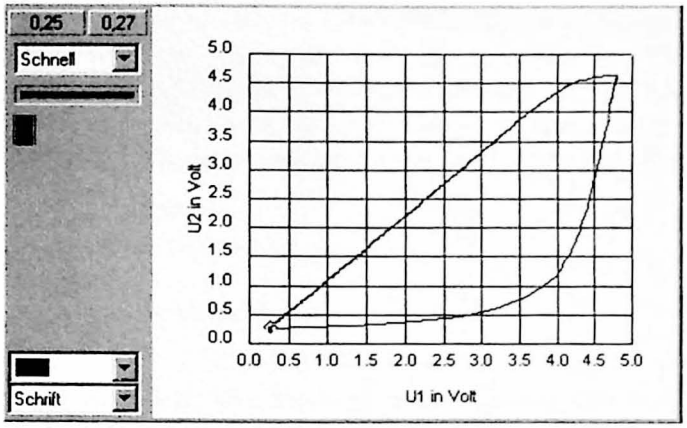


Abb. 8.4: Eine Messung mit dem XY-Shreiber

Neben den analogen Eingängen können auch die acht digitalen Eingänge überwacht werden. Der Bit-Schreiber ist ein digitaler Signal-Plotter für langsame Vorgänge.

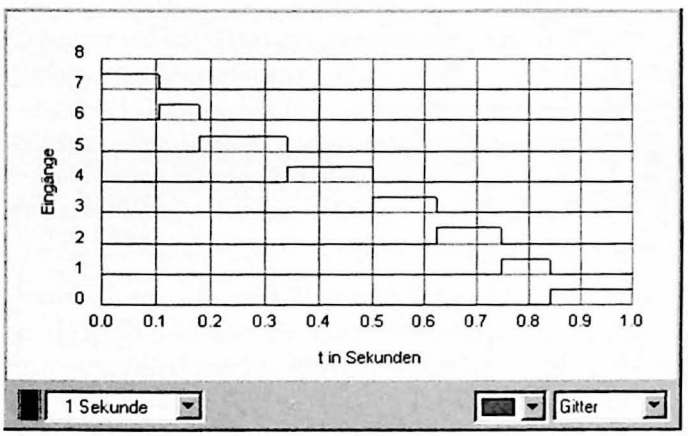


Abb. 8.5: Der Bit-Schreiber

Auch die hier gewonnenen Daten lassen sich kopieren und mit anderen Programmen weiterverarbeiten. Tabelle 8.2 zeigt typische digitale Daten als Text. Der Zustand des gesamten Eingangsports wird hier als ein Byte dargestellt.

Tabelle 8.2: Eingangszustände in Abhängigkeit von der Zeit

0,00	128
0,07	128
0,10	64
0,18	32
0,21	32
0,30	32
0,34	16
0,40	16
0,43	16
0,50	8
0,55	8

8.2 Die Programmierumgebung in Compact 2000

Compact 2000 enthält eine sehr einfache Programmierumgebung für kleinere Anwendungen. Die verwendete Syntax ist bereits aus dem Programm Do-It bekannt und wurde in erster Linie für die Ausbildung entwickelt. Schüler sollten ohne spezielle Programmiersprachen-Kenntnisse einfache Anwendungen realisieren und dabei die grundlegenden Prinzipen der Programmierung kennenlernen. Inzwischen wird die vereinfachte Programmierumgebung von Do-It auch im Hobbybereich und für einfache Aufgaben in der Industrie eingesetzt.

Man schreibt Programme durch Anklicken vorgegebener Schlüsselworte. Alle vorhandenen Befehle sind auf Registerkarten immer sichtbar, so dass man ohne spezielle Referenzen auskommt. Klickt man ein Schlüsselwort an, erscheinen weitere Registerkarten mit den möglichen Parametern. In dieser einfachen Form wird ein Programm ohne die Möglichkeit von Unterprogrammen geschrieben. Auch existiert nur eine einzige Variable „Zahl". Trotzdem lassen sich zahlreiche sinnvolle Programme schreiben. Einige Programmbeispiele finden sich auf der CD.

Vielfach wird das Programm für erste Versuche und Prototypen späterer Programme verwendet. Man kann oft mit geringem Aufwand die Hardware eines Projekts oder grundlegende Ansteuerungsverfahren überprüfen. Typische Aufgaben sind Signal-Generatoren, Ablaufsteuerungen, Regelkreise, Grenzwert-Überwachungen oder Testgeräte.

Eine häufige Aufgabe ist die Überwachung von Grenzwerten. Das Gerät soll zwei Spannungen messen und bei Überschreitung festgelegter Grenzwerte bestimmte

digitale Ausgänge setzen. Es wird eine Endlosschleife benötigt, die sich bei Bedarf abbrechen lässt. Nach der Wahl der Registerkarte „Wiederhole" erhält man eine Auswahl der möglichen Abbruchbedingungen einer Wiederholschleife. Eine Endlosschleife bricht man üblicherweise durch einen Tastendruck ab. Nach der Wahl der entsprechenden Bedingung erscheint ein äußerer Rahmen der Schleife:

```
Programm
  Wiederhole bis
  ...
  Tastendruck
Ende
```

In diese Schleife können dann weitere Befehle eingefügt werden. Für die gegebene Aufgabe wird eine Fallabfrage mit „Wenn" benötigt. *Abb. 8.6* zeigt das komplette Programm in der Programmierumgebung. Es werden nacheinander beide Analogeingänge abgefragt und mit den Grenzwerten 200 und 220 verglichen. Die Messwerte liegen in Form von Bytes, also Zahlen zwischen 0 (=0 V) und 255 (=5 V) vor. Eine Stufe entspricht damit etwa 20 mV. Die angegebenen Grenzwerte sind grob 4 V und 4,4 V. Eine oder mehrere Anweisungen nach „Wenn ..." werden ausgeführt, wenn die Bedingung wahr ist. Im anderen Fall werden Anweisungen nach „Sonst" ausgeführt. Hier werden jeweils die digitalen Ausgänge 0 und 1 eingeschaltet, wenn der Grenzwert überschritten ist und ausgeschaltet, wenn er nicht überschritten ist.

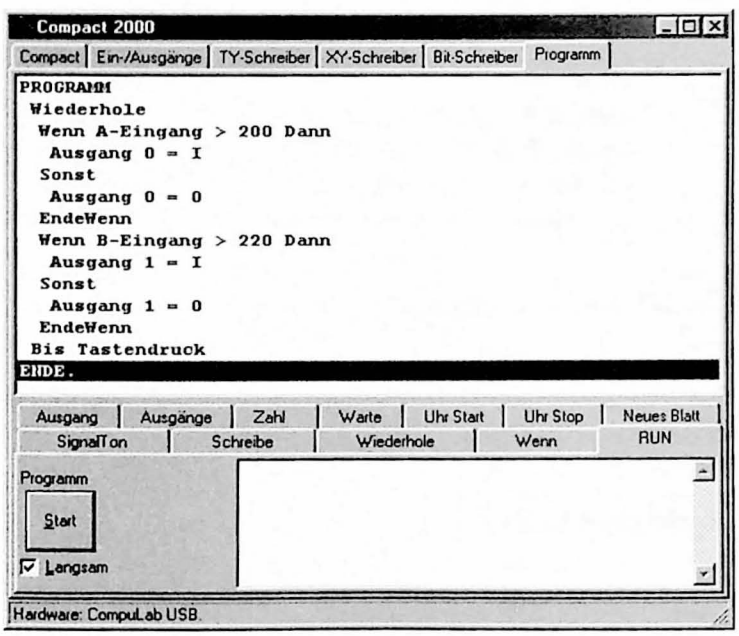


Abb. 8.6: Programmierung einer Grenzwert-Abfrage

Die Registerkarte RUN erlaubt den Start des Programms. Man hat die Wahl zwischen schneller und verzögerter Ausführung, bei der man den Ablauf des Programms direkt verfolgen kann.

Die Programmierumgebung unterstützt Multitasking. Ein gestartetes Programm läuft weiter, wenn man das Menü wechselt. Auf diese Weise kann man mit dem TY-Schreiber oder dem Bit-Schreiber Messungen an einem Versuch vornehmen, der durch Compact 200 selbst gesteuert wird. Auch kann ein Programm im Hintergrund seine Arbeit verrichten, wenn im Vordergrund andere Anwendungen laufen.

Die Programmierumgebung eignet sich für einfache Versuche und kleine Anwendungen. Der geringe Befehlsumfang begrenzt zwar die Möglichkeiten, erlaubt aber eine schnelle Einarbeitung und erleichtert die Orientierung. Die folgende Übersicht zeigt alle Befehle des Programms.

Ausgang
Verändert am Interface den Zustand eines der acht Digital-Ausgänge.

Parameter 1:

0 – 7:	Nummer des Ausgangs, der verändert werden soll.
Zufallswert:	Es wird ein zufälliger Ausgang gewählt.

Parameter 2:

O:	Der angegebene Ausgang wird ausgeschaltet.
I:	Der angegebene Ausgang wird eingeschaltet.
T:	Schaltet den Zustand dieses Ausgangs um.

Beispiel:

```
...
 Ausgang 0 = I
 Ausgang 1 = O
 Ausgang 2 = T
 Ausgang Zufallswert = T
...
```

Ausgänge
Verändert den Zustand der Digital-Ausgänge.

Parameter:
Dezimalzahl zwischen 0 und 255 oder ...

IOIOIOIO:	Verändert die Ausgänge nach dem angegebenen Muster.
Zahl:	Setzt die Ausgänge auf den Wert der Variablen Zahl.
Zufallswert:	Setzt alle Ausgänge zufällig.
Eingänge:	Die Ausgänge erhalten den Zustand der Eingänge,
A-Eingang:	erhalten den Wert des analogen A-Eingangs,
B-Eingang:	erhalten den Wert des analogen B-Eingangs.

Beispiel:

```
...
Ausgänge = IOIIIOIO
Ausgänge = 254
Ausgänge = Zahl
Ausgänge = A-Eingang
...
```

Zahl

Verändert die Variable Zahl.

Parameter 1:

=	Setzt die Variable gleich dem zweiten Parameter.
+, -	Addiert / subtrahiert den zweiten Parameter.
*, /	Multipliziert / dividiert den zweiten Parameter.

Parameter 2:
Dezimalzahl zwischen 0 und 255 oder...

A-Eingang:	Der Wert des analogen A-Eingangs.
B-Eingang:	Der Wert des analogen B-Eingangs.
Zufallswert:	Ein Zufallswert zwischen 0 und 255.
Eingänge:	Der binäre Wert der 8 Eingänge.

Hinweis:
Wenn durch die Operation die Grenzen von 0 bzw. 255 überschritten werden, erfolgt ein Übertrag auf Null.

Beispiel:

```
...
Zahl = 255
Zahl – IOIOIOIO
Schreibe Zahl
Zahl * A-Eingang
Zahl / B-Eingang
Schreibe Zahl
```

Warte
Wartet die angegebene Zeit ab.

Parameter: Zeitdauer in Sekunden.

Beispiel:

```
...
Schreibe „Dies sind 10 Sekunden..."
Warte 10 Sekunden
Schreibe „... und das nur 0.5 ... „
Warte 0.5 Sekunden
Schreibe „...Sekunden !"
...
```

Uhr Start
Setzt die Variable Zeit auf Null und startet die interne Uhr.

Parameter: Keine.

Hinweis:
Nach und während dem Befehl „Uhr Start" kann der Stand der internen Uhr mit dem Befehl „Schreibe Zeit" ausgegeben werden.

Beispiel:

```
...
Uhr Start
Wiederhole
      Schreibe „Interne Uhr ist :" Zeit
Bis Tastendruck
Uhr Stop
Schreibe „Interne Uhr ist stehengeblieben bei" Zeit
...
```

Uhr Stop
Stoppt die interne Uhr.

Parameter: Keine.

Hinweis:
Die Zeit kann mit dem Befehl „Schreibe Zeit" ausgegeben werden.

Beispiel:

```
...
Uhr Start
Wiederhole
      Schreibe „Interne Uhr ist :“ Zeit
Bis Tastendruck
Uhr Stop
Schreibe „Interne Uhr ist stehengeblieben bei“ Zeit
...
```

Neues Blatt
Löscht das Ausgabefenster, das vom Befehl „Schreibe“ benutzt wird.

Parameter: Keine.

Beispiel:

```
...
Schreibe „Dieser Text ist nur kurz zu sehen...“
Warte 2 Sekunden
Neues Blatt
Schreibe „...fertig.“
...
```

Schreibe
Gibt die Parameter im Ausgabefenster aus.

Parameter:

Zeit	Stand der internen Uhr in Sekunden.
A-Eingang	Wert des A – Einganges.
B-Eingang	Wert des B – Einganges.
Zufallswert	Ein Zufallswert zwischen 0 und 255.
Eingänge	Der binäre Zustand der Eingänge 0 – 7.
Zahl	Der Wert der Variablen Zahl.
„ Text “	Ein beliebiger Text in Anführungszeichen.

Beispiel:

```
...
Uhr Start
Schreibe „A-Eingang ist „A-Eingang
Schreibe „B-Eingang ist „B-Eingang
Schreibe „Eine Zufallszahl „Zufallswert
Schreibe „Eingänge sind „Eingänge“ bei Zeit = „Zeit
Uhr Stop
```

SignalTon
Gibt einen kurzen Ton über den PC-Lautsprecher aus.

Parameter: Keine.

Beispiel:

```
...
SignalTon
Schreibe „Achtung!“
...
```

Wiederhole / Bis [Bedingung]
Die zwischen „Wiederhole“ und „Bis [Bedingung]“ stehenden Befehle werden solange wiederholt, bis während deren Bearbeitung der Zustand [Bedingung] eingetreten ist. Dabei kann größer als, kleiner als und gleich als Vergleichsoperand benutzt werden. Die Vergleichsmöglichkeiten hängen vom gewählten Parameter ab.

Bedingungen:

Tastendruck	Die Schleife wird nach einem Tastendruck verlassen,
Durchläufe	nach einer bestimmten Zahl von Durchläufen,
Zeit ...	Sekundenvor/bei/nach einer bestimmten Zeit,
Zahl	bis Zahl gleich, größer, kleiner dem angegebenen Wert,
A-Eingang	bis A gleich, größer, kleiner dem angegebenen Wert,
B-Eingang	bis B gleich, größer, kleiner dem angegebenen Wert,
Eingänge	bis Eingänge gleich, größer, kleiner ...
Eingang	bis ein bestimmter Eingang I oder O ist.

Beispiel:

```
...
Wiederhole
Schreibe „drücke eine Taste...“
Bis Tastendruck
...
```

Wenn [Bedingung] Dann ... Sonst ... EndeWenn
Im Gegensatz zur „Wiederhole ... Bis“ – Schleife wird hier eine Bedingung geprüft und je nach Ergebnis eine Folge von Befehlen abgearbeitet.

Bedingungen:

Tastendruck	Die Bedingung ist ein erfolgter Tastendruck,
Durchläufe	wenn der Durchlaufszähler einen bestimmten Wert hat,
Zeit ... Sekunden	wenn Zeit gleich, größer, kleiner ...
Zahl	wenn Zahl gleich, größer, kleiner ...
A-Eingang	wenn A gleich, größer, kleiner ...
B-Eingang	wenn B gleich, größer, kleiner ...
Eingänge	wenn Eingänge gleich, größer, kleiner ...
Eingang	wenn ein bestimmter Eingang I oder O ist.

Beispiel:

```
...
Wenn Eingang 1 = I Dann
      Schreibe „Ja“
Sonst
      Schreibe „Nein“
EndeWenn
```

9 Der USB-Controller AN2131

Die Firma ANCHOR Chips (inzwischen übernommen von Cypress, www.cypress.com) stellt einen 8051-kompatiblen Mikrocontroller mit internem USB-Kern unter der Bezeichnung EZ-USB her. EZ steht für Easy und betont den einfachen Umgang mit dem System. Neben einem erweiterten 8051-Kern enthält der Chip auch internes RAM und das komplette USB-Interface. Der Controller hat die Typenbezeichnung AN2131S im 44-poligen Gehäuse bzw. AN2131Q im 80-poligen Gehäuse. Die verwendeten PQFP-Gehäuse sind allein für SMD-Montage vorgesehen. Leider gibt es keine Typen im DIL-Gehäuse. Die Betriebsspannung des Chips beträgt 3,3 V, 5-V-Betrieb ist nicht möglich. Die geringere Betriebsspannung ermöglicht eine einfache Versorgung über den USB, dessen 5-V-Spannung herabstabilisiert werden kann.

9.1 Technische Daten

Der EZ-USB-Kern ist ein verbesserter 8051 mit erheblich schnellerer Programmausführung. Der Kern wird als DW8051 bezeichnet und ähnelt dem 80C320 von Dallas. Ein Befehlszyklus dauert nicht wie üblich 12 Quarztakte, sondern nur vier (vgl. *Abb. 9.1*). Die Quarzfrequenz von 12 MHz wird durch eine interne PLL verdoppelt. Bei einer Taktfrequenz von 24 MHz ergibt sich ein Befehlszyklus von 166,7 ns. Damit ist der Prozessor sechsmal schneller als ein Standard-8051 mit 12 MHz. Während allerdings beim 8051 fast alle Befehle mit einem oder zwei Befehlszyklen auskommen, verwendet der DW8051 viele Befehle mit drei oder vier Zyklen, was den Geschwindigkeitsvorsprung zum Teil wieder reduziert.

Die im EZ-USB verwendeten Ports weichen stark von normalen 8051-Ports ab. Es handelt sich hier nicht um quasi-bidirektionale Ports mit Pullup, sondern um echte bidirektionale Ports mit einer Umschaltung der Datenrichtung. In Ausgaberichtung erhält man CMOS-Gegentakt-Ausgänge, in Eingaberichtung sind die Ports hochohmig. Alle Ports sind 5-V-kompatibel, so dass 5-V-Chips in der Schaltung verwendet werden können.

Ein normaler 8051-Prozessor steuert seine Ports über Spezialfunktionsregister im internen Adressraum 128...255. Der EZ-USB dagegen kennt die normalen Portzugriffe des 8051 nicht, sondern verwendet völlig eigene Ports, die über Register im externen

Adressraum angesprochen werden, wie sonst extern adressierte Peripheriebausteine. Das „externe" RAM befindet sich bis zu einer Größe von 8 K mit auf dem Chip, darüber hinaus kann extern weiters RAM bis zu 64 K angeschlossen werden.

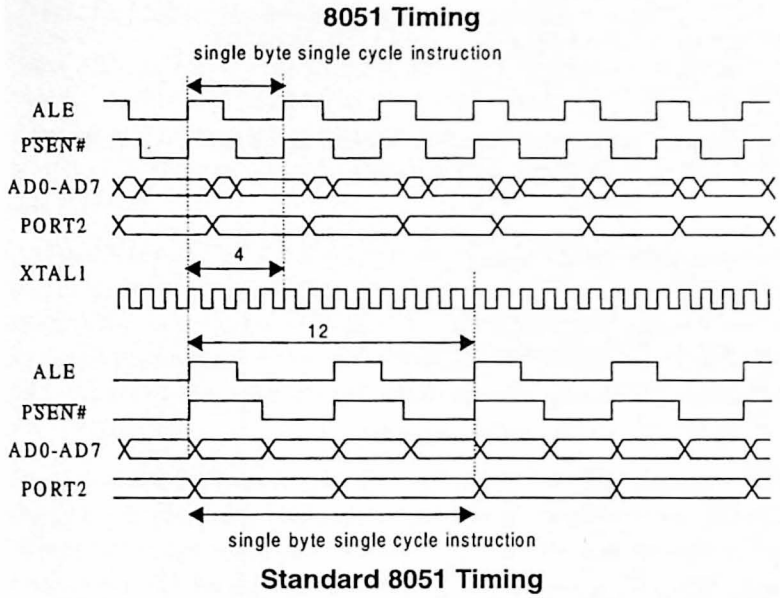


Abb. 9.1: Das verbesserte Timing des DW-8051-Kerns (Anchor Chips)

Tabelle 9.1: Vergleichende Daten im Überblick

Feature	Intel				Dallas DS80C320	Anchor 8051
	8031	8051	80C32	80C52		
Clocks per instruction cycle	12	12	12	12	4	4
Program / Data Memory	-	4 KB ROM	-	8 KB ROM	-	8 K RAM
Internal RAM	128 bytes	128 bytes	256 bytes	256 bytes	256 bytes	256 bytes
Data Pointers	1	1	1	1	2	2
Serial Ports	1	1	1	1	2	2
16-bit Timers	2	2	3	3	3	3
Interrupt sources (total of int. and ext.)	5	5	6	6	13	13
Stretch memory cycles	no	no	no	no	yes	yes

Das 8-K XRAM auf dem Chip erfüllt mehrere Funktionen:

- Es beherbergt Programme, die in den EZ-USB geladen werden
- Es dient als allgemeines Daten-RAM
- Es stellt die Verbindung zwischen Prozessor und USB-Kern her
- Es enthält bestimmte Steuerregister

Die folgenden Steuerregister im Adressraum des XRAM ermöglichen die Portzugriffe und den Zugriff auf die Reset-Leitung des Prozessors.

Tabelle 9.2: Die Port-Register des AN2131

7F92	CPUCS	Zugriff auf den Prozessor-Reset
7F93	PORTACFG	Configurationsregister A
7F94	PORTBCFG	Configurationsregister B
7F95	PORTCCFG	Configurationsregister C
7F96	OUTA	Port A Ausgaben
7F97	OUTB	Port B Ausgaben
7F98	OUTC	Port C Ausgaben
7F99	PINSA	Port A Eingaben
7F9A	PINSB	Port B Eingaben
7F9B	PINSC	Port C Eingaben
7F9C	OEA	Datenrichtung Port A
7F9D	OEB	Datenrichtung Port B
7F9E	OEC	Datenrichtung Port C

Jeder Portanschluss kann zwei Funktionen annehmen, eine Portfunktion oder eine Spezialfunktion des Prozessors. Nach einem Reset und nach dem Neueinschalten des Systems enthalten alle Konfigurationsregister den Wert 0. Alle Anschlüsse haben also die normale Portfunktion, solange nicht etwas anders eingestellt wird.

Als Port kann jeder Anschluss ein Eingang oder ein Ausgang sein. Die Datenrichtung wird über die OE-Register festgelegt. Nach einem Reset enthalten alle den Wert 0, so dass sie Eingänge sind. Der Zustand der Eingänge kann über die zugehörigen PINS-Register abgefragt werden. Dies ist übrigens möglich, ohne dass der Prozessor ein Programm dazu benötigt, da alle Register sowohl vom Prozessor als auch über den USB abgefragt werden können.

Portausgaben erfordern zunächst eine Umschaltung der entsprechenden Bits in den OE-Registern. Dann kann über das entsprechende OUT-Register der Portzustand festgelegt werden. Beides ist nur vom Prozessor aus möglich, direkte Zugriffe über den USB sind leider gesperrt. Um über den USB auf Ausgänge zuzugreifen, muss daher ein kleines Interface-Programm eingesetzt werden.

Das CPUCS-Register ist dagegen direkt über den USB zugänglich. Schreibt man den Wert 1 in dieses Register, dann wird die Reset-Leitung des Prozessors gesetzt. Mit dem Wert 0 wird der Reset zurückgenommen, so dass ein Programm im Prozessor startet. Ein Programm kann zuvor ab Adresse 0000 über den USB in das RAM geladen werden.

Auf der Basis des EZ-USB-Chips sollen hier verschiedene Interfaces entwickelt werden. Der Prozessor ermöglicht sehr einfache Schaltungen. Man benötigt nicht viel mehr als einen Quarz, ein paar Widerstände und einige Kondensatoren. Je nach Aufgabe können die Ports direkt eingesetzt werden oder externe Elektronik wie z.B. AD- oder DA-Wandler ansteuern. Neben den Portleitungen steht auch ein I²C-Bus zur Verfügung.

Meist hat die Softwareentwicklung zwei Seiten. Zum einen muss ein 8051-Steuerprogramm für die jeweilige Aufgabe entwickelt werden. Zum anderen muss ein Windows-Programm geschrieben werden, das das Interface initialisiert und den Datenverkehr ausführt. Die Initialisierung besteht darin, das 8051-Interfaceprogramm in den Chip zu übertragen und zu starten. Der dann folgende Datenverkehr kann frei definiert werden, da er zwischen den beiden eigenen Programmen durchgeführt wird. Zwischen den beiden Softwareebenen liegt der USB-Kern im EZ-USB, das USB-Kabel, das USB-Interface im PC und ein USB-Treiber.

9.2 Das EZ-USB-Starterkit

Zum AN2131 gibt es ein umfassendes Starterkit (vgl. *Abb. 9.2*). Die Platine enthält neben dem Prozessor noch 64 K RAM und umfangreiche Peripherie wie z.B. zwei serielle Schnittstellen und mehrere Bausteine am I²C-Bus. Neben einer vollständigen Dokumentation enthält das Kit umfangreiche Beispielsoftware. Der Anwender enthält Einblicke in alle Ebenen der Software-Entwicklung von der Firmware über die Anwendersoftware auf dem Host-System bis zu Treibern. Für alle Treiber des Systems liegen auch die Quelltexte vor.

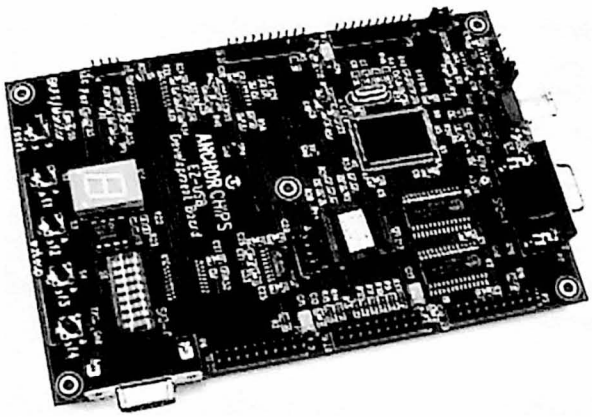


Abb. 9.2: Die Starterkit-Platine zum AN2131

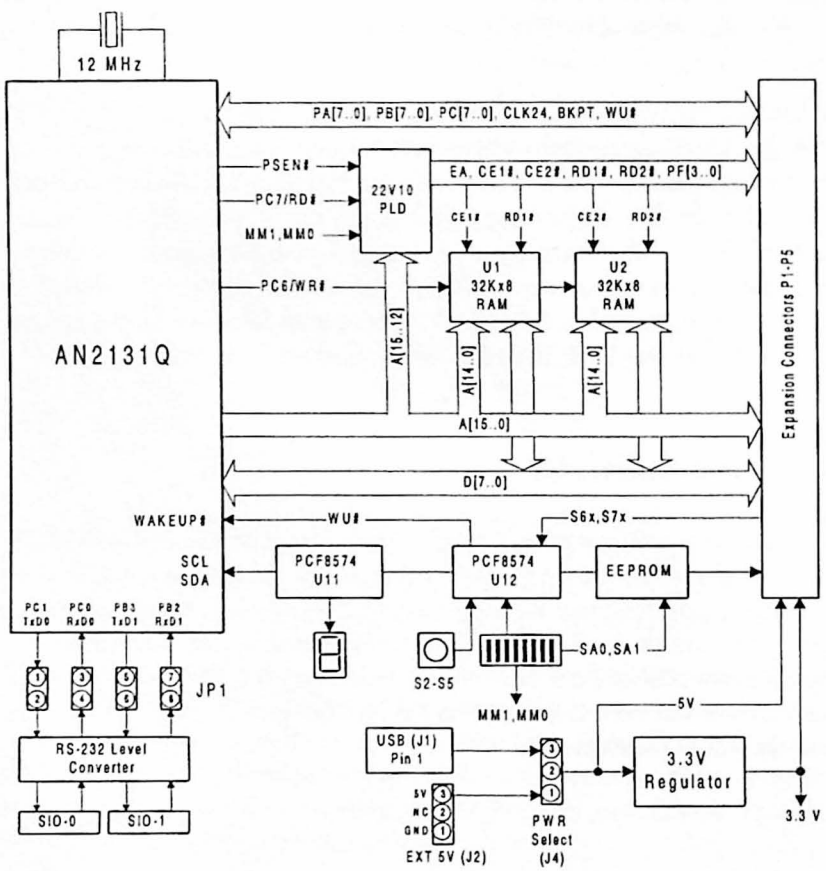


Abb. 9.3: Blockschaltbild der Starterkit-Platine

Das Starterkit wird mit umfangreicher Software geliefert. Neben einem Mehrzweck-USB-Treiber für das EZ-USB-System werden zahlreiche Beispiele im Quelltext und als fertige Anwendungen vorgestellt. Mit enthalten ist ein C-Compiler der Firma Keil mit einer maximalen Kodegröße von 2 Kilobytes. Das EZ-USB Control-Panel bietet eine wirksame Hilfe bei der Einarbeitung in die USB-Problematik und bei der Entwicklung. Der Anwender kann beliebige Requests generieren und die Antwort des USB-Geräts beobachten. *Abb. 9.4* zeigt die Abfrage des Device-Deskriptors.

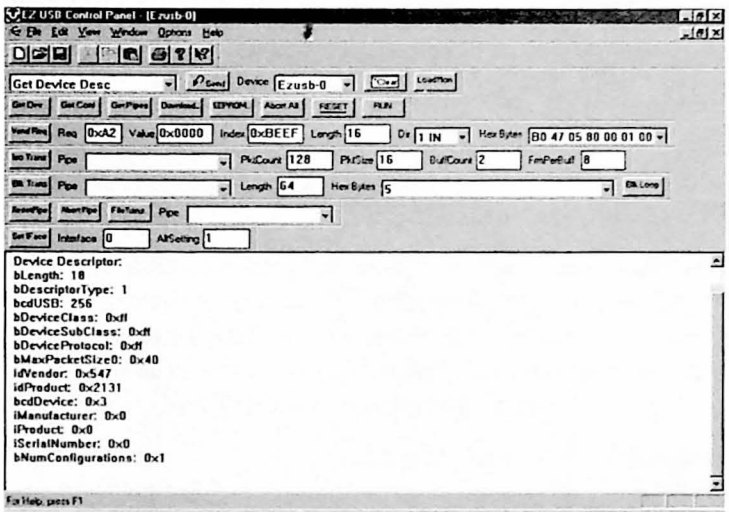


Abb. 9.4: Abrufen des Device Deskriptors im Control-Panel

9.3 Das Default Anchor Device

Der EZ-USB-Chip ist bereits ohne eine spezielle Firmware betriebsbereit und stellt ein vollständiges USB-Gerät dar, das als „Default Anchor Device" bezeichnet wird. Der USB-Kern hält alle notwendigen Eigenschaften und Deskriptoren für eine eigenständige Enumeration bereit. Das angemeldete USB-Gerät ist in der Lage, Software in das RAM des 8051-Prozessors zu laden und den Prozessor zu starten. Außerdem können Daten aus dem RAM gelesen werden. Damit kann man mit geringstem Aufwand kleine USB-Anwendungen entwickeln. Sogar ohne ein geladenes Programm kann das EZ-USB-Gerät bereits Portabfragen über den USB durchführen.

Das Default-Gerät besteht aus einer einzelnen USB-Konfiguration (Interface 0) mit drei Alternate-Settings. Alternate Setting 0 stellt nur den Control-Endpoint 0 bereit. Die Einstellungen 1 und 2 liefern außerdem einen Interrupt-Endpoint, sechs Bulk-

Endpoints und sechs isochrone Endpoints. Sie unterscheiden sich in der erforderlichen Bus-Bandbreite der Interrupt- und isochronen Endpoints.

Tabelle 9.3: Transfer-Typen und FIFO-Größen der Endpoints im Anchor Default Device

Endpoint	Typ	Alternative Setting 0	Alternative Setting 1	Alternative Setting 2
0	CTL	64	64	64
1 IN	INT	0	16	64
2 IN	Bulk	0	64	64
2 OUT	Bulk	0	64	64
3 IN	Bulk	0	64	64
2 OUT	Bulk	0	64	64
4 IN	Bulk	0	64	64
2 OUT	Bulk	0	64	64
8 IN	ISO	0	16	256
8 OUT	ISO	0	16	256
9 IN	ISO	0	16	16
9 OUT	ISO	0	16	16
10 IN	ISO	0	16	16
10 OUT	ISO	0	16	16

Nach der Enumeration befindet sich das Gerät im Alternate-Setting 0, d.h. es können nur Control-Transfers durchgeführt werden. Dies eröffnet aber schon sehr viele Möglichkeiten, die in den folgenden Kapiteln besprochen werden.

Will man spezielle Firmware für andere Transfer-Arten wie z.B. den Bulk-Transfer nutzen, dann muss die Alternate-Setting 1 oder 2 eingeschaltet werden.

Der Anwender hat auch die Möglichkeit, eigene Deskriptoren zu verwenden. Im einfachsten Fall wird nur die Vendor-ID, Produkt-ID und die Device-ID aus einem angeschlossenen seriellen EEPROM geladen und ausgetauscht. Der EZ-USB-Chip meldet sich weiterhin mit allen Einstellungen des Anchor Default Device, aber mit eigenen Hersteller-Angaben.

Zusätzlich bietet der Chip die Möglichkeit, völlig eigene Deskriptoren einzusetzen. Der USB-Kern enthält dazu ein Steuerbit „ReNum", das bestimmt, ob der USB-Kern oder der 8051-Controller die Standard-USB-Requests beantwortet. Die Voreinstellung für das ReNum-Bit ist Null, wobei der USB-Kern alle Requests beantwortet. Man kann nun eine vollständige USB-Firmware ins RAM laden und starten.

Der 8051-Controller setzt dann das ReNum-Bit. Das Gerät meldet sich vom Bus ab, indem es hardwaremäßig eine Trennung simuliert. Danach erfolgt eine neue Anmeldung mit den eigenen Deskriptoren.

Alternativ zum Download einer Software über den USB kann sie auch beim Start des Systems aus einem angeschlossenen seriellen EEPROM geladen werden.

Der EZ-USB-Chip bietet flexible Möglichkeiten der Entwicklung unterschiedlicher USB-Geräte. Besonders für Einsteiger ist es hilfreich, dass man mit minimalen Firmware-Programmen bereits sinnvolle Anwendungen erzeugen kann. Auf diese Weise kann ein Anwender die eigentliche USB-Problemtik zunächst fast vollständig ausblenden.

9.4 Treiberaufrufe in Delphi

Die Firma Anchor-Chips (jetzt Cypress) stellt einen universellen Treiber EZUSB.SYS für Windows98 zum Zugriff auf den EZ-USB-Chip bereit. Der Treiber unterstützt alle wichtigen Zugriffe, so dass darauf verzichtet werden kann, einen eigenen Gerätetreiber zu entwickeln. Von Windows aus werden nur drei Funktionen benötigt, um mit dem Treiber zu kommunizieren:

CreateFile()	Öffnen des Kommunikationskanals
DeviceIoControl()	Ausführen der Treiberfunktionen
CloseHandle()	Schließen des Kanals

Um einen ersten Kontakt zum Interface zu bekommen, bietet es sich an, den Divice-Deskriptor abzufragen. Das folgende kleine Programm in Delphi zeigt, wie es geht. Der Chip antwortet ohne jede zusätzliche Software auf die ersten Anfragen des Systems, was die Entwicklung eigener Applikationen erheblich vereinfacht. In einfacheren Systemen muss bereits ein Programm des Mikrocontrollers dafür sorgen, dass der Chip sich ordentlich anmeldet.

Abb. 9.5: Abfrage von ID-Nummern aus dem Device-Deskriptor

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    procedure Button1Click(Sender: TObject);
  private
    { Private-Deklarationen}
  public
    { Public-Deklarationen}
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

type _USB_DEVICE_DESCRIPTOR = record
    bLength: UCHAR;
    bDescriptorType: UCHAR;
    bcdUSB: WORD;
    bDeviceClass: UCHAR;
    bDeviceSubClass: UCHAR;
    bDeviceProtocol: UCHAR;
    bMaxPacketSize0: UCHAR;
    idVendor: WORD;
    idProduct: WORD;
    bcdDevice: WORD;
    iManufacturer: UCHAR;
    iProduct: UCHAR;
    iSerialNumber: UCHAR;
    bNumConfigurations: UCHAR;
end;

procedure TForm1.Button1Click(Sender: TObject);
var TemplateHandle: THandle;
    bresult: Boolean;
    nBytes: DWord;
    PnBytes: ^DWord;
    DeviceHandle: THandle;
    USB_DEVICE_DESCRIPTOR: _USB_DEVICE_DESCRIPTOR;
```

```
begin
  PnBytes := @nBytes;
  DeviceHandle := CreateFile ('\\.\ezusb-0',
 Generic_write,File_Share_write,nil,open_existing,0,TemplateHandle);
  bResult :=
DeviceIoControl(DeviceHandle,$00222004,nil,0,@USB_DEVICE_DESCRIPTOR,18,nBytes,nil);
  Edit1.Text := 'VID: ' + FloatToStr (USB_DEVICE_DESCRIPTOR.idVendor);
  Edit2.Text := 'PID: ' + FloatToStr (USB_DEVICE_DESCRIPTOR.idProduct);
  CloseHandle (DeviceHandle);
end;

end.
```

Listing 9.1: Auslesen des Device-Deskriptors

Der Device-Deskriptor ist eine allgemein definierte Struktur mit einer Länge von exakt 18 Bytes, die für jedes Gerät gilt. Sie wird hier in einer Typdefininition aufgebaut. Für den eigentlichen Zugriff muss zuerst mit CreateFile ein Handle auf den Treiber geholt werden. Der Treiber heißt für den ersten EZ-USB-Chip am USB „ezusb-0". Falls ein weiterer Chip am Bus vorhanden wäre, müsste dessen Treiber-Instanz mit „ezusb-1" angesprochen werden. Mit dem erhaltenen Handle führt DiviceIoControl den eigentlichen Treiberzugriff aus und überträgt den Device-Descripter aus dem ES-USB. Die beiden gewünschten Elemente des Descriptors, nämlich VID und PID werden dann in Textfenstern dargestellt. Am Ende wird der Treiberzugriff mit CloseHandle wieder geschlossen.

Über die DeviceIoControl-Funktion wird dem EZUSB-Treiber eine Funktionsnummer übergeben, über die die gewünschte Treiberfunktion gewählt wird. Hier lautet diese Funktionsnummer $00222004 und bedeutet, dass der Device-Deskriptor abgefragt werden soll. Es existieren zahlreiche weitere Treiberfunktionen, die alle nach einem genau definierten Muster aufgebaut sind. Sie werden weiter unten noch genauer erläutert.

10 Ein Fullspeed-USB-Interface

Hier wird ein einfaches USB-Interface mit dem AN2131 vorgestellt. Es dient sowohl für allgemeine Anwendungen im Bereich Messen, Steuern und Regeln als auch als kleines Entwicklungssystem für USB-Anwendungen. Im Gegensatz zum weiter oben vorgestellten CompuLAB-USB handelt es sich hier um ein Fullspeed-System.

Das Gerät ist geeignet für Portzugriffe, I²C-Bus-Anwendungen und die Ansteuerung eines AD-Wandlers MAX186 mit 8 Kanälen und 12 Bit. Es kann zur Programmentwicklung verwendet werden, zur Einarbeitung in die USB-Technik, als reines Analoginterface oder als universelles Interface für digitale Ein- und Ausgaben.

10.1 Schaltung und Aufbau

Die Platine enthält neben dem Prozessor nur relativ wenig zusätzliche Elektronik. Außer dem Spannungsregler für 3,3 V gibt es zwei Sockel für einen AD-Wandler und für ein serielles EEPROM. Beide ICs können wahlweise eingesetzt werden. Der AD-Wandler erweitert das Gerät zu einem 12-Bit-Analoginterface mit acht Eingangskanälen. Das EEPROM dient zur Bereitstellung von ID-Nummern zur Enumerierung mit anderen als den Anchor-Treibern.

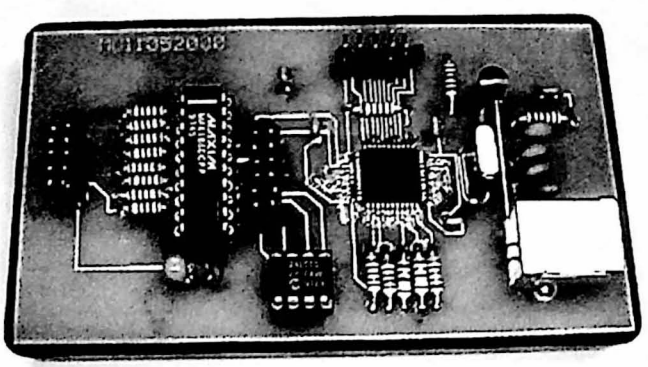


Abb. 10.1: Ein Prototyp des Geräts

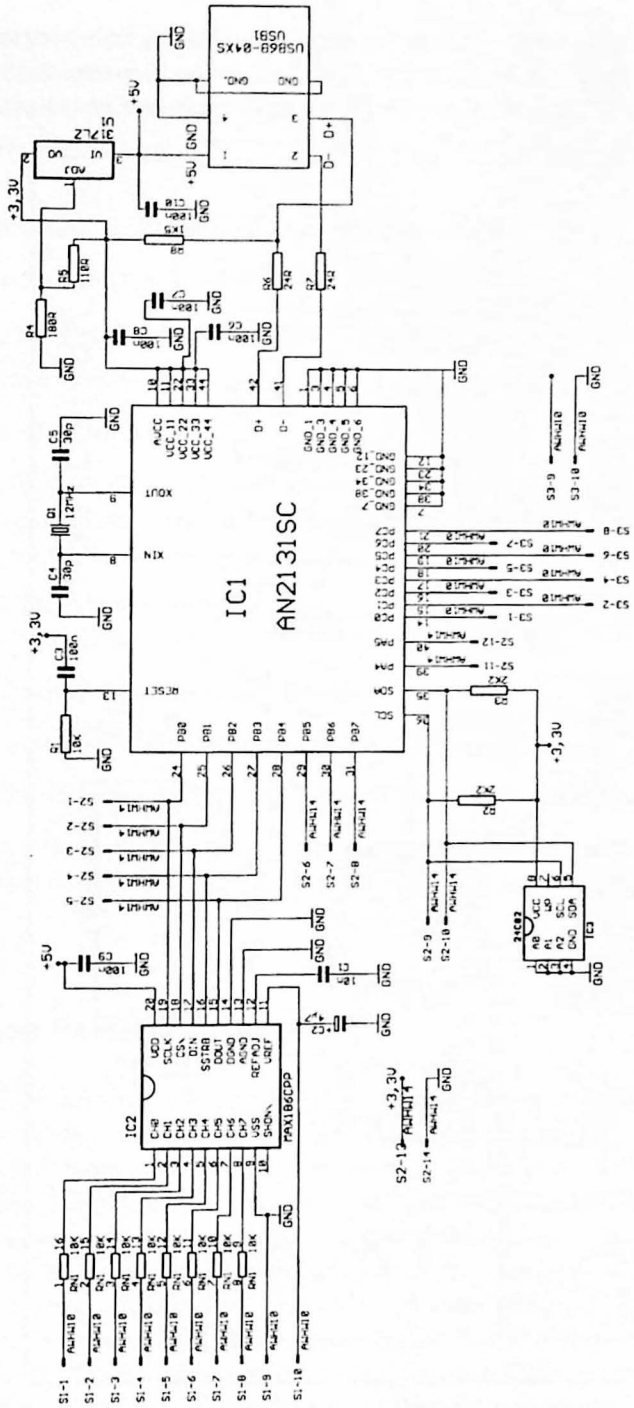


Abb. 10.2: Schaltbild des USB-Interfaces

Das Gerät kann unter der Bezeichnung Serai-USB (Serielles Input/Output System) fertig aufgebaut von der Firma Modul-Bus bezogen werden. In der Anwendung als Analog-Interface wird es von einer passenden Software unterstützt.

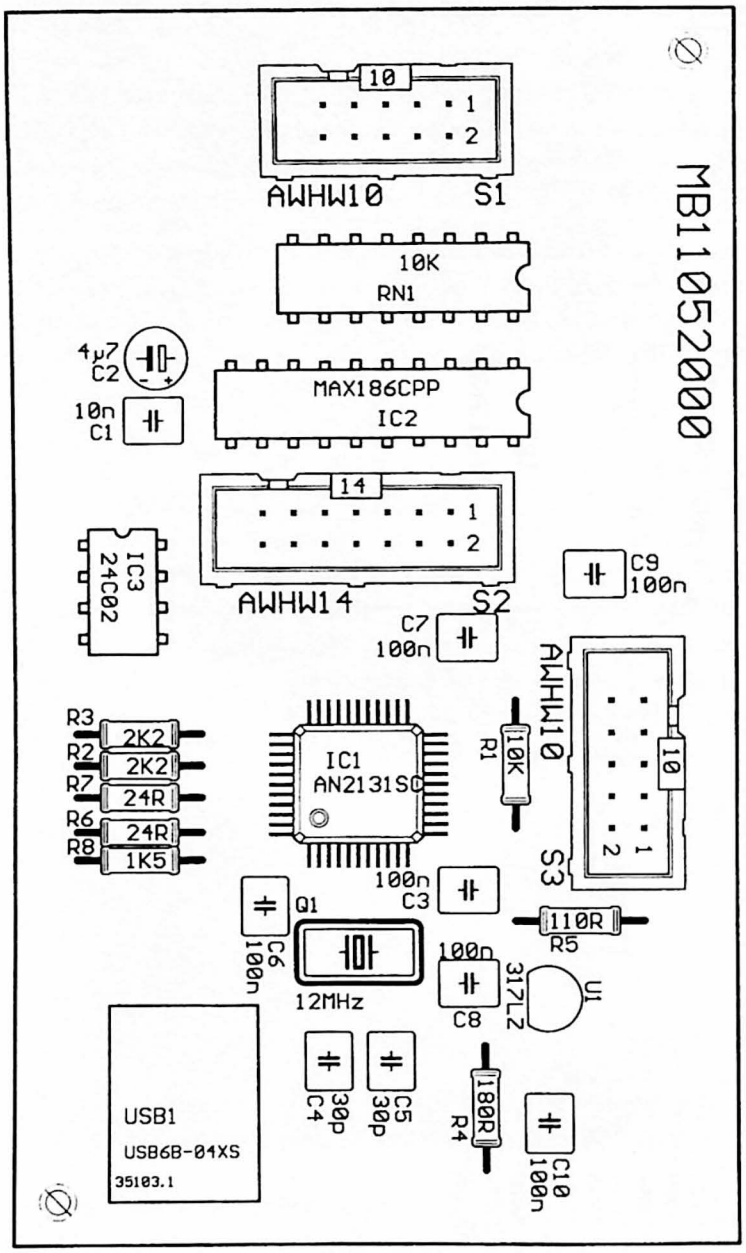


Abb. 10.3: Das Platinen-Layout

Das Interface verwendet einen AN2131S im 44-poligen PQFP-Gehäuse. Das kleine Gehäuse verzichtet gegenüber der 80-poligen Version auf einige Portanschlüsse und den gesamten externen Adress- und Datenbus. Das ist jedoch zu verschmerzen, wenn man bedenkt, dass der Chip bereits 8 Kilobyte internes Programm- und Daten-RAM hat.

Folgende Anschlüsse sind an Pfostensteckern herausgeführt:

- Zwei Anschlüsse von Port A, PA4 und PA5
- Der komplette Port B mit acht Anschlüssen
- Der komplette Port C mit acht Anschlüssen
- Der I²C-Bus mit SDA und SCL

Auf der Platine befindet sich ein Sockel für den AD-Wandler MAX186. Wenn er bestückt wird, fallen die fünf niederwertigen Anschlüsse von Port B weg. Dafür erhält man acht hochauflösende Analogeingänge.

Denkbar ist eine andere Verwendung der Ports, die etwa die gleichen Möglichkeiten wie das CompuLAB-USB bietet:

- Port B wird ein kompletter 8-Bit-Eingangsport
- Port C wird ein kompletter 8-Bit-Ausgangsport
- Über den I²C-Bus wird ein PCF8591 als AD/DA-Wandler angeschlossen

Im folgenden sollen Portzugriffe und die notwendigen Steuerprogramme erläutert werden. Das System arbeitet dabei ohne EEPROM mit dem Anchor-EZUSB-Treiber.

10.2 Lesen von Portzuständen

Die Ports des EZ-USB können ohne ein eigenes Controllerprogramm direkt über den USB gelesen werden, indem man die PINS-Register im Adressbereich ab 7F99 ausliest. Der EZUSB-Treiber stellt für allgemeine Zugriffe auf den Adressraum eine Funktion VENDOR_REQUEST_IN mit der Funktionsnummer $00222014 zur Verfügung. Damit lassen sich beliebige Adressen auslesen und einzelne Bytes schreiben. Das Windows-Programm muss eine Datenstruktur vom Typ _VENDOR_REQUEST_IN übergeben, die als neuer Typ deklariert wird. Darin enthalten ist ein Eintrag bRequest der mit dem Wert $A0 die gewünschte Zugriffsfunktion auf dem Adressbereich bewirkt. Der Anwender muss zusätzlich einen Puffer ausreichender Größe an den Treiber übergeben, in dem die eigentlichen Daten übergeben werden.

Zusammen mit dem Funktionsaufruf wird auch die gewünschte Datenrichtung übergeben. Beim Auslesen können Blöcke bis zu 1024 Bytes in einem Stück übertragen werden. Verwendet man dieselbe Funktion zum Schreiben von Daten, dann kann jeweils nur ein Byte übergeben werden.

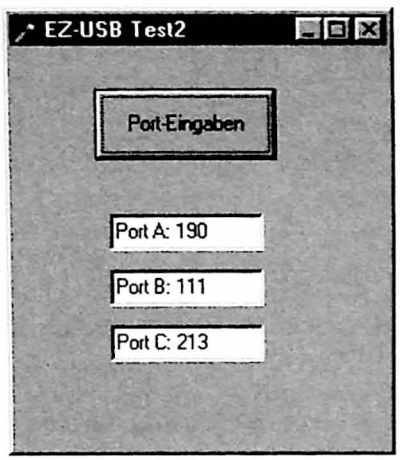


Abb. 10.4: Abfrage der Ports

```
unit EZUSB2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    procedure Button1Click(Sender: TObject);
  private
    { Private-Deklarationen}
  public
    { Public-Deklarationen}
  end;

var
  Form1: TForm1;

implementation
```

```
{$R *.DFM}
type USHORT = WORD;

type _VENDOR_REQUEST_IN = record
    bRequest : Byte;
    wValue : Word;
    wIndex : Word;
    wLength: Word;
    direction : Byte;
    bData : Byte;
end;

procedure TForm1.Button1Click(Sender: TObject);
var TemplateHandle: THandle;
    DeviceHandle: THandle;    bresult: Boolean;
    nBytes: DWord;
    MyRequest: _VENDOR_REQUEST_IN;
    Puffer:Array [1..1024]of Byte;
    n: Integer;
begin
   myRequest.bRequest := $A0;
   myRequest.wValue := $7F99;     //Startadresse
   myRequest.wIndex := 0;         //nicht gebraucht
   myRequest.wLength := $03;      //max. 1024
   myRequest.direction := 1;      //Auslesen
   myRequest.bData := $00;     { wenn Length=1 dann setzt
                                 der Treiber dieses als Daten ein.}
  //Verwendet IOCTRL_ezusb_Vendor_Request  00222014
  //Geht nur für Ausgabe eines Einzelbyte
  //oder zum Auslesen von Datenbereichen
  //aber nicht zum Download von Blöcken
  DeviceHandle := CreateFile ('\\.\ezusb-0',
Generic_write,File_Share_write,nil,open_existing,0,
TemplateHandle);
  bResult := DeviceIoControl(DeviceHandle,$00222014,
   @myRequest,10,@Puffer,SizeOf(Puffer),nBytes,nil);
  if bResult then Edit1.Text := 'Port A: '+
     FloatToStr(Puffer[1]);
  if bResult then Edit2.Text := 'Port B: '+
     FloatToStr(Puffer[2]);
  if bResult then Edit3.Text := 'Port C: '+
     FloatToStr(Puffer[3]);
  CloseHandle (DeviceHandle);
end;

end.
```

Listing 10.1: Auslesen von Portzuständen

Im Programmbeispiel werden drei Bytes ab Adresse $7F99 ausgelesen. Damit erhält man die Zustände der Register PINSA, PINSB und PINSC. Da nach einem Reset alle Ports als Eingänge geschaltet sind, lassen sich alle Portzustände ohne weitere Vorbereitung direkt lesen.

Die Treiberfunktion lässt auch ein direktes Schreiben von Bytes in den Adressraum des Prozessors zu. Wie weiter unten noch gezeigt wird, lässt sich der Prozessor-Reset auf diese Weise steuern. Auch die Übertragung einzelner Bytes in das XRAM ist ohne weiters möglich. Versucht man jedoch, die Portzustände direkt zu verändern, zeigt sich, dass die relevanten Register für Schreibzugriffe über den USB gesperrt sind. Sie können nur vom Prozessor aus verändert werden. Dazu ist ein kleines Programm nötig, wie im folgenden gezeigt wird.

10.3 Portausgaben

Portausgaben über den EZ-USB erfordern ein kleines Maschinenprogramm im Mikrocontroller. Es hat die Aufgabe, einen Port in Ausgaberichtung zu schalten und dann entsprechende Ausgaben weiterzuleiten. Speziell für den Download von Datenbereichen und Programmen gibt es die Treiberfunktion ANCHOR_DOWNLOAD mit der Funktionsnummer $0022206D. Es können beliebig lange Codesequenzen übertragen werden. Dem Treiber wird ein Puffer mit dem Programmcode übergeben. Zusätzlich wird die Struktur ANCHOR_DOWNLOAD_CONTROL übergeben, die nur die Startadresse enthält.

Das eigentliche Maschinenprogramm ist sehr kurz. Es kopiert zunächst den Wert $FF in das Register OEC an der Adresse $7F9E. Damit wird der gesamte Port C in Ausgaberichtung geschaltet. Dann kopiert es einen Startwert (hier $00) in das Register OUTC an der Adresse $7F98. Es folgt ein Sprung, mit dem die letzte Ausgabe endlos wiederholt wird. Über den USB kann dann durch Übertragung eines einzelnen Bytes an die richtige Adresse das Programm selbst modifiziert werden, so dass ein anderer Ausgabewert verarbeitet wird.

Das gesamte Maschinenprogramm wird hier direkt in das Delphi-Programm geschrieben, indem die einzelnen Bytes dem Ausgabepuffer zugewiesen werden. Zur Verdeutlichung der einzelnen Abläufe wird der Initialisierungsvorgang auf eine eigene Schaltfläche gelegt. Nach der Übertragung muss das Maschinenprogramm zunächst gestartet werden.

Der Start des Programms wird eingeleitet, indem der Rest-Zustand des Prozessors aufgehoben wird. Über den USB wird dazu der Wert Null in das CPUCS-Register an der Adresse $7F92 geschrieben. Umgekehrt kann mit dem Wert 1 ein Reset aus-

gelöst und damit das Programm gestoppt werden. Beide Funktionen werden hier über eigene Schaltflächen ausgelöst. Zur Übertragung des einzelnen Bytes könnte zwar ebenfalls die Funktion ANCHOR_DOWNLOAD verwendet werden. Genauso gut geht es aber mit der schon weiter oben verwendeten Funktion VENDOR_REQUEST.

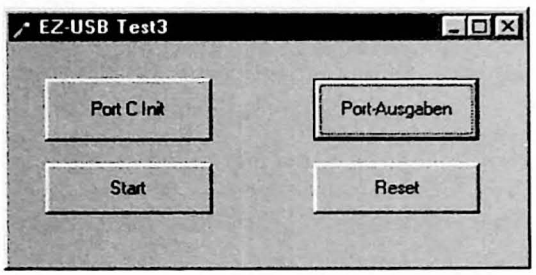


Abb. 10.5: Portausgaben

```
unit EZUSB3;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Ausgaben: TButton;
    PortInit: TButton;
    Start: TButton;
    Reset: TButton;
    procedure AusgabenClick(Sender: TObject);
    procedure PortInitClick(Sender: TObject);
    procedure StartClick(Sender: TObject);
    procedure ResetClick(Sender: TObject);
  private
    { Private-Deklarationen}
  public
    { Public-Deklarationen}
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
type _ANCHOR_DOWNLOAD_CONTROL = record
   Offset: Word;
```

```
end;

type _VENDOR_REQUEST_IN = record
    bRequest : Byte;
    wValue : Word;
    wIndex : Word;
    wLength: Word;
    direction : Byte;
    bData : Byte;
end;


procedure TForm1.PortInitClick(Sender: TObject);
var TemplateHandle: THandle;
    DeviceHandle: THandle;
    bresult: Boolean;
    nBytes: DWord;
    downloadControl: _ANCHOR_DOWNLOAD_CONTROL;  //mit Startadresse
    Puffer:Array [1..15]of Byte; {Beliebig viele mit ANCHOR-DOWNLOAD}
begin
  Puffer[1]:=$90;   // mov DPTR,$7F9E
  Puffer[2]:=$7F;
  Puffer[3]:=$9E;
  Puffer[4]:=$74;    // mov A,#FF
  Puffer[5]:=$FF;
  Puffer[6]:=$F0;    // mov A,@DPTR
  Puffer[7]:=$90;    // mov DPTR,$7F98
  Puffer[8]:=$7F;
  Puffer[9]:=$98;
  Puffer[10]:=$74;   // mov A,$00
  Puffer[11]:=$00;   // Ausgabebyte an Adr. 000A
  Puffer[12]:=$F0;   // mov @DPTR,A
  Puffer[13]:=$02;   // LJMP $0009
  Puffer[14]:=$00;
  Puffer[15]:=$09;
  downloadControl.offset:=0;
  DeviceHandle := CreateFile ('\\.\ezusb-0',Generic_write,File_Share_write,nil,
       open_existing,0,TemplateHandle);
  bResult := DeviceIoControl(DeviceHandle,$0022206D,@downloadControl,
       sizeof(downloadControl), @Puffer,sizeof(Puffer),nBytes,nil);
  CloseHandle (DeviceHandle);
end;

procedure TForm1.StartClick(Sender: TObject);
var TemplateHandle: THandle;
    DeviceHandle: THandle;
    bresult: Boolean;
    nBytes: DWord;
    MyRequest: _VENDOR_REQUEST_IN;
begin
   myRequest.bRequest := $A0;
```

```
   myRequest.wValue := $7F92;
   myRequest.wIndex := 0;
   myRequest.wLength := 1;
   myRequest.direction := 0;
   myRequest.bData := 0;      {Reset=0}
   DeviceHandle := CreateFile ('\\.\ezusb-0',Generic_write,File_Share_write,nil,
        open_existing,0,TemplateHandle);
   bResult := DeviceIoControl(DeviceHandle,$00222014,@myRequest,10,nil,
        0,nBytes,nil);
   CloseHandle (DeviceHandle);
end;

procedure TForm1.ResetClick(Sender: TObject);
var TemplateHandle: THandle;
    DeviceHandle: THandle;
    bresult: Boolean;
    nBytes: DWord;
    MyRequest: _VENDOR_REQUEST_IN;
begin
   myRequest.bRequest := $A0;
   myRequest.wValue := $7F92;
   myRequest.wIndex := 0;
   myRequest.wLength := 1;
   myRequest.direction := 0;
   myRequest.bData := 1;      {Reset=1}
   DeviceHandle := CreateFile ('\\.\ezusb-0',Generic_write,File_Share_write,nil,
        open_existing,0,TemplateHandle);
   bResult := DeviceIoControl(DeviceHandle,$00222014,@myRequest,10,nil,
        0,nBytes,nil);
   CloseHandle (DeviceHandle);
end;

procedure TForm1.AusgabenClick(Sender: TObject);
var TemplateHandle: THandle;
    DeviceHandle: THandle;    bresult: Boolean;
    nBytes: DWord;
    MyRequest: _VENDOR_REQUEST_IN;
    Puffer:Array [1..1024]of Byte;
    n: Integer;
begin
   myRequest.bRequest := $A0;
   myRequest.wValue := $000A;     //Startadresse
   myRequest.wIndex := 0;         //nicht gebraucht
   myRequest.wLength := $01;      //max. 1024
   myRequest.direction := 0;      //Schreiben
   myRequest.bData := $00;     { wenn Length=1 dann setzt der Treiber dieses
                                 als Daten ein.}

  //Verwendet IOCTRL_ezusb_Vendor_Request  00222014
  //Geht nur für Ausgabe eines Einzelbyte
  //oder zum Auslesen von Datenbereichen
```

```
  //aber nicht zum Download von Blöcken
  DeviceHandle := CreateFile ('\\.\ezusb-0',Generic_write,File_Share_write,nil,
        open_existing,0,TemplateHandle);
  for n:= 0 to 255 do begin
    myRequest.bData := n;
    bResult := DeviceIoControl(DeviceHandle,$00222014,@myRequest,10,nil,
          0,nBytes,nil);
  end;
  CloseHandle (DeviceHandle);
end;

end.
```

Listing 10.2: Portausgaben über ein nachgeladenes Programm

Bei der Erprobung des Programms muss zuerst die Initialisierung mit dem Download des Maschinenprogramms ausgeführt werden. Dann ist das Programm zu starten. Und schließlich können Ausgaben vorgenommen werden. Die Ausgabeprozedur erzeugt aufsteigende Zahlenfolgen, die sich als Ausgabemuster an den Anschlüssen des Ports C beobachten lassen. Die schnellsten Zustandswechsel am Portanschluss PC0 zeigen einen Wechsel nach jeweils 4 ms. Dies ist also die Reaktionszeit der Ausgaben.

Mit der Reset-Schalfläche kann man das Programm stoppen. Die Ausgaben über den USB funktionieren weiterhin, führen aber zu keiner Portänderung, weil das Maschinenprogramm nicht aktiv ist. Übrigens kann auch das Maschinenprogramm immer wieder neu übertragen werden. Dazu ist es nicht einmal nötig, den Prozessor zu stoppen. Das XRAM des EZ-USB funktioniert wie ein Dual-Ported-RAM, das von zwei Seiten aus beschrieben und gelesen werden kann. Der Prozessor „merkt" nichts davon, dass der USB auf sein RAM zugreift. Nach einer vollständigen Ausgabeschleife ist der Zustand aller Portleitungen 1, da als letztes der Wert 255 übertragen wurde. Überträgt man dann das Programm bei laufendem Prozessor neu, wechseln alle Leitungen in den 0-Zustand, weil dies der Startwert bei der Initialisierung ist.

10.4 Basisfunktionen für USB-Zugriffe

Mit den bisherigen Programmen hat man fast alle Werkzeuge zum Aufbau beliebiger Interfaces. Ganz ähnlich wie bei einfachen Portausgaben kann man auch bei der Ansteuerung beliebiger Peripherie wie z.B. eines AD-Wandlers vorgehen. Es wird jeweils ein kleines Controllerprogramm gebraucht, das die eigentliche Peripherie ansteuert. Es muss in den EZ-USB übertragen und gestartet werden. Der eigentliche

Datenaustausch zwischen dem Controller und dem USB findet im einfachsten Fall immer direkt über das RAM statt. Dazu muss nur eine Adresse oder ein Datenfeld vereinbart werden, über das Daten ausgetauscht werden können. Damit die Arbeit etwas einfacher wird, sollten einige Grundfunktionen in einer eigenen Unit zusammengefasst werden:

- Download aus einer Binärdatei oder Intel-Hex-Datei
- Start/Reset-Funktion
- Beliebige Schreibzugriffe auf einzelne Byteadressen
- Beliebige Schreibzugriffe auf Datenbereiche
- Beliebige Lesezugriffe auf den Datenbereich

Die folgende Delphi-Unit liefert alle erforderlichen Zugriffe auf das 8051-System. Speicherbereiche können gelesen oder beschrieben werden. Mit dem Zugriff auf einzelne Adressen wird auch der Start und Reset eines Programms möglich.

```
unit EZUSB;

interface
uses Windows;

type _ANCHOR_DOWNLOAD_CONTROL = record
   Offset: Word;
end;

type _VENDOR_REQUEST_IN = record
    bRequest : Byte;
    wValue : Word;
    wIndex : Word;
    wLength: Word;
    direction : Byte;
    bData : Byte;
end;

var  InBuffer: Array [0..8191]of Byte;
     OutBuffer: Array [0..8191]of Byte;
procedure WrRAM(Adr: Word; Dat: Byte);
function RdRAM(Adr: Word): Byte;
procedure WriteRAMbytes(StartAdr, NumberOfBytes: Word);
procedure ReadRAMbytes (StartAdr, NumberOfBytes: Word);
procedure ProgStart();
procedure ProgReset();
procedure DownloadBin(FileName:  String);
procedure DownloadIntelHex (FileName: String);
```

```
implementation
procedure WrRAM(Adr: Word; Dat: Byte);        //single Byte
var TemplateHandle: THandle;
    DeviceHandle: THandle;
    bresult: Boolean;
    nBytes: DWord;
    MyRequest: _VENDOR_REQUEST_IN;
begin
   myRequest.bRequest := $A0;
   myRequest.wValue := Adr;
   myRequest.wIndex := 0;
   myRequest.wLength := 1;
   myRequest.direction := 0;
   myRequest.bData := Dat;
   DeviceHandle := CreateFile ('\\.\ezusb-0',Generic_write,
      File_Share_write,nil,open_existing,0,TemplateHandle);
   bResult := DeviceIoControl(DeviceHandle,$00222014,
      @myRequest,10,nil,
   0,nBytes,nil);
   CloseHandle (DeviceHandle);
end;

function RdRAM(Adr: Word): Byte;
var TemplateHandle: THandle;
    DeviceHandle: THandle;    bresult: Boolean;
    nBytes: DWord;
    MyRequest: _VENDOR_REQUEST_IN;
    n: Integer;
    Buffer: Array [1..2] of Byte;
begin
   myRequest.bRequest := $A0;
   myRequest.wValue := Adr;
   myRequest.wIndex := 0;
   myRequest.wLength := 1;
   myRequest.direction := 1;     // Read
   myRequest.bData := $00;

  DeviceHandle := CreateFile ('\\.\ezusb-0', Generic_write,
     File_Share_write,nil,open_existing,0,TemplateHandle);
  bResult := DeviceIoControl(DeviceHandle,$00222014,
     @myRequest,10,@Buffer,SizeOf(Buffer),nBytes,nil);
  CloseHandle (DeviceHandle);
  RdRAM := Buffer[1];
end;

procedure WriteRAMbytes(StartAdr, NumberOfBytes: Word);
var TemplateHandle: THandle;
    DeviceHandle: THandle;
    bresult: Boolean;
```

```
    nBytes: DWord;
    downloadControl: _ANCHOR_DOWNLOAD_CONTROL;
begin
  downloadControl.offset:=StartAdr;
  DeviceHandle := CreateFile ('\\.\ezusb-0',Generic_write,
     File_Share_write,nil,open_existing,0,TemplateHandle);
  bResult := DeviceIoControl(DeviceHandle,
     $0022206D,@downloadControl,sizeof(downloadControl),@OutBuffer,
     NumberOfBytes,nBytes,nil);CloseHandle (DeviceHandle);
end;

procedure ReadRAMbytes (StartAdr, NumberOfBytes: Word);
var TemplateHandle: THandle;
    DeviceHandle: THandle;    bresult: Boolean;
    nBytes: DWord;
    MyRequest: _VENDOR_REQUEST_IN;
    n: Integer;
begin
   myRequest.bRequest := $A0;
   myRequest.wValue := StartAdr;
   myRequest.wIndex := 0;
   myRequest.wLength := NumberOfBytes; //max. 1024
   myRequest.direction := 1;     // Read
   myRequest.bData := $00;
  DeviceHandle := CreateFile ('\\.\ezusb-0',Generic_write,
     File_Share_write,nil,open_existing,0,TemplateHandle);
  bResult := DeviceIoControl(DeviceHandle,$00222014,
     @myRequest,10,@InBuffer,SizeOf(InBuffer),nBytes,nil);
  CloseHandle (DeviceHandle);
end;


procedure ProgStart();
begin
   WrRAM($7F92,0);
end;

procedure ProgReset();
begin
   WrRAM($7F92,1);
end;

function HexToInt (Hex: String): Byte;
var h, l: Byte;
begin
  h:= ord (Hex[1])-48;
  if h>9 then h:=h-7;
  l:= ord (Hex[2])-48;
  if l>9 then l:=l-7;
```

```
  HexToInt := 16*h+l;
end;

procedure DownloadBin(FileName:  String);
vAR  f :File of Byte;
     r :Byte;
     Adresse, MaxAdresse, n, i : Word;
     Code: Byte;
begin
  AssignFile(f,FileName);
  {$I-} Reset(f); {$I+}
  r:=IOResult;
  if r = 0 then begin
    Adresse := 0;
    for n:=1 to FileSize(f)do begin;
      Read(f,Code);
      OutBuffer [Adresse] := Code;
      Adresse := Adresse + 1;
    end;
    CloseFile(f);
    WriteRAMbytes (0,Adresse)
  end;
end;

procedure DownloadIntelHex (FileName: String);
 VAR  f :TextFile;
      Line: String;
      r,wert :Byte;
      Num: Byte;
      Adr, MaxAdr, n, i, Code : Word;
begin
    AssignFile(f,FileName);
    {$I-} Reset(f); {$I+}
    r:=IOResult;
    IF r = 0 then begin
      MaxAdr :=0;
      repeat
        Readln(f,line);
       Num := HexToInt (copy(line,2,2));
        Adr := 256*HexToInt(copy(line,4,2))
               +HexToInt(copy(line,6,2));
        if Num>0 then begin
          for i := 1 to Num do begin
             code := HexToInt(copy(line,8+2*i,2));
             OutBuffer[Adr] := code;
             if Adr > MaxAdr then MaxAdr := Adr;
             Adr := Adr + 1;
          end;
        end;
```

```
      until Num = 0;
       CloseFile(f);
     end;
     WriteRAMbytes (0,MaxAdr+1);
  end;

end.
```

Listing 10.3: Unit EZUSB zum Zugriff auf RAM-Bereiche

Die Unit enthält u.a. Prozeduren zum Download von Programmdateien im Intel-Hex-Format und im Binärformat. Die folgenden Anwendungen demonstrieren den Gebrauch.

10.5 Ein Logik-Analysator

Die hohe Übertragungsrate des USB ermöglicht die Realisierung eines digitalen Speicherscopes oder Logikanalysators. Digitale Zustände werden über einen Port eingelesen und im RAM zwischengespeichert. Speicherblöcke werden dann über den USB ausgelesen und am PC dargestellt. Das folgende Assemblerprogramm zeigt eine einfache Schleife zur Aufnahme von Daten vom Port C. Der Speicher wird ab Adresse 1024 gefüllt. Ein eigentliches Ende des Bereichs ist nicht festgelegt. Das Programm muss daher vom PC aus gestartet und nach wenigen Millisekunden wieder gestoppt werden.

Die eigentliche Speicherschleife muss möglichst kurz und schnell sein, damit eine gute zeitliche Auflösung erreicht wird. Bei einer Taktfrequenz von 24 MHz ergibt sich ein Befehlszyklus von 166,7ns. Damit erreicht das Programm nach Listing 10.5 eine Abtastrate von 150 kHz.

```
;EZUSB, Logik1.asm
#include 8051.h

OEC  .equ   7F9Eh
PINSC .equ  7F9Bh
OUTC .equ   7F98h
     mov DPTR,#OEC          ;Port C Input Enable
     mov A,#00h
     movx @DPTR,A

     mov DPTR,#1024         ;RAM-Start
Loop Push DPL               ;(2)
     Push DPH               ;(2)
```

```
    mov DPTR,#PINSC          ;(3)
    movx A,@DPTR             ;(2)
    Pop  DPH                 ;(2)
    POP  DPL                 ;(2)
    movx @DPTR,A             ;(2)
    inc DPTR                 ;(2)
    SJMP Loop                ;(3)
    .end                     ;(20)
```

Listing 10.4: Aufnahme digitaler Zustände

Anders als bei einem üblichen 8051-Prozessor wird der Port nicht über ein SFR im internen RAM-Bereich erreicht, sondern über ein Register im externen RAM-Bereich. Der Datapointer muss also abwechselnd auf die Portadresse und auf die aktuelle Speicheradresse zeigen. Das Programm legt den DPTR zwischenzeitlich immer wieder auf dem Stack ab. Insgesamt werden für eine Programmschleife 20 Zyklen benötigt. Daraus ergibt sich eine Abtastrate von exakt 150 kHz.

Die Abtastrate lässt sich auf 250 kHz erhöhen, wenn man den zweiten Datapointer des DW8051 einsetzt. Das SFR DPS (86h) entscheidet, welcher der beiden vorhandenen Pointer aktuell genutzt wird. Da nur Bit 0 ausgewertet wird, reicht ein INC-Befehl mit nur einem Taktzyklus, um zwischen DPTR0 und DPTR1 zu wechseln. Man spart auf diesen Weise das Zwischenspeichern des DPTR auf dem Stack.

```
;EZUSB, Logik2.asm, using two Data Pointers
#include 8051.h

OEC  .equ   7F9Eh
PINSC .equ  7F9Bh
OUTC .equ   7F98h
DPS  .equ   86h

     mov DPTR,#OEC   ;Port C Input Enable
     mov A,#00h
     movx @DPTR,A

     mov DPS,#0  ;select DPTR0
     mov DPTR,#1024
     mov DPS,#1  ;select DPTR1
     mov DPTR,#PINSC
     mov DPS,#0  ;select DPTR0

Loop inc DPS   ;select DPTR1  ;(1)
     movx A,@DPTR             ;(2)
     inc DPS  ;select DPTR0   ;(1)
     movx @DPTR,A             ;(2)
```

```
    inc DPTR                ;(3)
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
SJMP Loop          ;(3)
    .end
```

Listing 10.5: Verwendung beider Datapointer

Das Assemblerprogramm wird mit zusätzlichen NOP-Befehlen so verlangsamt, dass sich genau die gewünschte Abtastrate von 250 kHz ergibt.

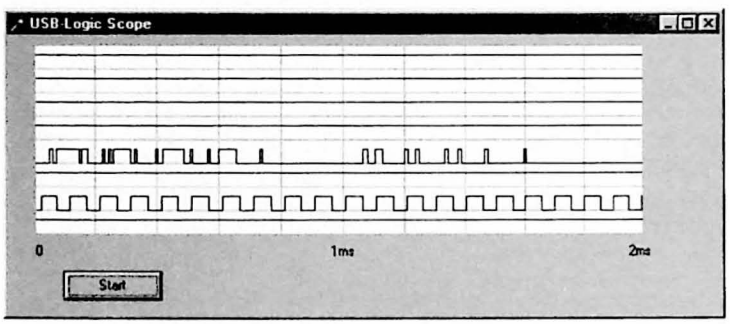


Abb. 10.6: Eine Beispielmessung mit dem Logik-Analysator

Die Darstellung der Daten (vgl. *Abb. 10.6*) liefert die Zustände aller acht Eingangskanäle über insgesamt 2 Millisekunden. Das Delphi-Programm nach Listing 10.6 zeigt das Auslesen und Darstellen der aufgezeichneten digitalen Daten.

```
unit Logik;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls, EZUSB;

type
  TForm1 = class(TForm)
    PaintBox1: TPaintBox;
    Start: TButton;
```

```
    procedure FormCreate(Sender: TObject);
    procedure StartClick(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}


procedure TForm1.FormCreate(Sender: TObject);
begin
  Paintbox1.Canvas.Pen.Color:=ClBlack;
  Paintbox1.Canvas.Brush.Color:=ClWhite;
  PaintBox1.Canvas.Rectangle(0,0,500,160);
  ProgReset;
  DownLoadIntelHex ('Logik2.Obj');
end;

procedure TForm1.StartClick(Sender: TObject);
var n, m, Wert: Integer;
begin
  ProgStart;
  //Sleep (20);    //keine Wartezeit, 4 ms reicht
  ProgReset;
  ReadRamBytes (1024,1010);
  ReadRamBytes (0,100);
  Paintbox1.Canvas.Pen.Color:=ClLtGray;
  Paintbox1.Canvas.Brush.Color:=ClWhite;
  PaintBox1.Canvas.Rectangle(0,0,501,161);
  for n:= 1 to 8 do begin
      paintbox1.canvas.MoveTo(0,n*20);
      Paintbox1.Canvas.Lineto (500,n*20);
  end;
  for n:= 1 to 10 do begin
      paintbox1.canvas.MoveTo(n*50,0);
      Paintbox1.Canvas.Lineto (n*50, 160);
  end;
  Paintbox1.Canvas.Pen.Color:=ClBlack;
  Wert := 1;
  for n := 1 to 8 do begin
    for m := 2 to 1000 do begin
      paintbox1.canvas.MoveTo ((m-1)div 2 ,n*20-12*
            ((InBuffer[m-1] and Wert) div Wert));
      Paintbox1.Canvas.Lineto  ((m-1)div 2  ,#
            n*20-12*((InBuffer[m] and Wert) div Wert));
      Paintbox1.Canvas.Lineto  (m div 2 ,n*20-12*
            ((InBuffer[m] and Wert) div Wert));
    end;
```

```
    Wert := Wert * 2;
  end;

end;

end.
```

Listing 10.6: Delphi-Programm zur Darstellung der gemessenen Daten

11 Der AD-Wandler MAX186

Der MAX186 kann optional auf dem oben vorgestellten Fullspeed-USB-Interface eingesetzt werden. Hier werden zunächst die Anschlüsse und die erforderliche Ansteuerung beschrieben. Ein fertig einsetzbares Anwenderprogramm ermöglicht den Betrieb als Messgerät. Zu weitergehenden Anwendungen werden Beispiele für die erforderliche Firmware und Anwenderprogramme in Delphi vorgestellt.

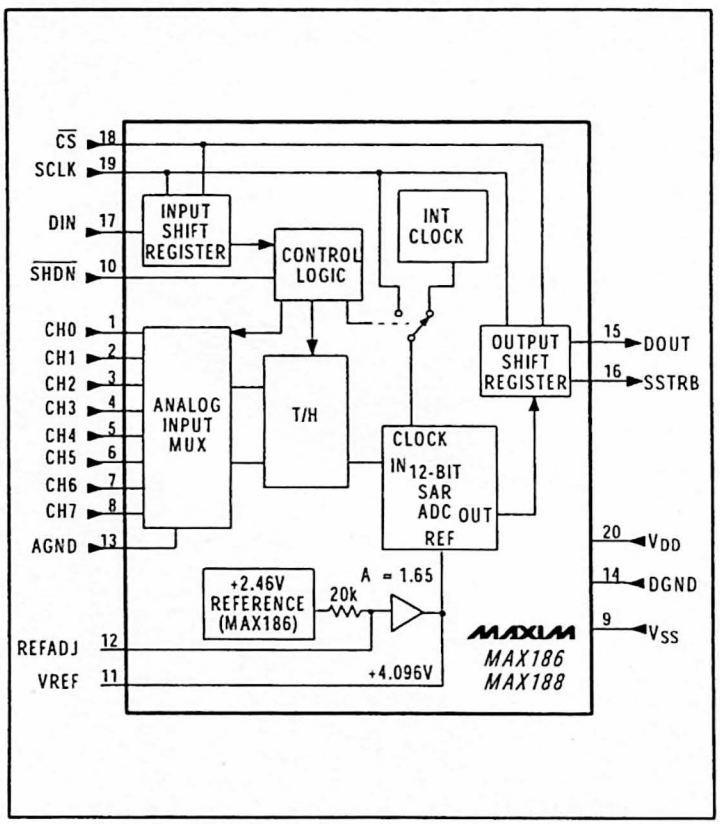


Abb. 11.1: Blockschaltbild des MAX186 (Maxim)

11.1 Anschlüsse und Betriebsarten

Der MAX186 ist ein 12-Bit AD-Wandler in CMOS-Technik mit einem 8-Kanal-Multiplexer und einem seriellen SPI-Anschluss mit vier Leitungen zum direkten Anschluss an einen Mikrocontroller bei Taktraten bis zu 4 MHz. Der Wandler arbeitet mit einfacher 5-V-Versorgung oder mit +/- 5V. Die Analogeingänge können unipolar oder bipolar arbeiten. Der Wandler besitzt eine interne Referenz mit 4,096 V. Es werden zwei Stromsparmodi unterstützt, so dass man den Verbrauch bis auf 10 µA reduzieren kann.

Tabelle 11.1: Anschlüsse des MAX186

Pin	Bezeichnung	Funktion
1	CH0	Eingangskanal 0
	bis	
8	CH7	Eingangskanal 7
9	AGND	Masse des Analogteils
10	/SHDN	Low: Power-down-Modus (10µA)
11	VREF	Referenzspannung 4,096V
12	REFADJ	Justiereingang der Referenz
13	AGND	Masse des Analogteils
14	DGND	Masse des Digitalteils
15	DOUT	Serieller Datenausgang
16	SSTRB	Statusausgang für laufende Wandlung
17	DIN	Serieller Dateneingang
18	/CS	Chip-Selekt-Eingang
19	SCLK	Serieller Clock-Eingang
20	VDD	Versorgung +5V

Der Wandler kennt zahlreiche Betriebsarten, die beim Start einer Wandlung über das seriell übertragene Steuerbyte festgelegt werden. Die Wandlung kann mit internem oder externem Takt erfolgen. Wählt man den externen Takt, dann erfolgt die eigentliche Wandlung mit ihrer sukzessiven Approximation synchron zum Auslesevorgang. Dabei muss eine symmetrische Rechteckform eingehalten werden, und es darf eine Taktrate von 100 kHz nicht unterschritten werden, weil es sonst durch Entladung des internen Abtast-Halteglieds zu einer Verschlechterung der Messgenauigkeit kommen kann. Hier wird nur mit internem Takt gearbeitet.

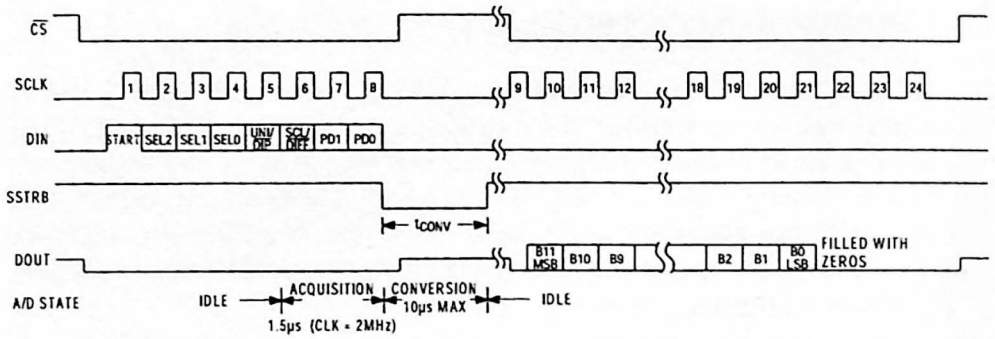


Abb. 11.2: Impulsdiagramm der Ansteuerung mit internem Takt

Die Betriebsarten werden über zwei Bits des Steuerbytes eingestellt. Es stehen zwei Power-down-Modi zur Verfügung, die vor allem bei batteriebetriebenen Geräten von Bedeutung sind. Weitere Steuerbits wählen die Art der Messung aus. Außer den üblichen Masse-bezogenen Eingängen können Differenzeingänge gewählt werden, wobei immer zwei Eingänge zusammenarbeiten. Neben der unipolaren Messung (0...4,095V) ist für Differenzeingänge auch eine bipolare Betriebsart (-2,047V ... +2,047V, bezogen auf den negativen Eingang) möglich. Die aktiven Eingangskanäle werden über drei Bits (SEL0...SEL2) gewählt.

Die Übertragung des Steuerworts beginnt immer mit dem als Startbit gesetzten MSB. Tabelle 11.2 zeigt den Aufbau des Steuerbytes:

Tabelle 11.2: Aufbau des Steuerbytes

Bit	Name	Funktion
7	START	immer gesetzt
6	SEL2,	
5	SEL1,	Auswahl des Eingangskanals
4	SEL0	
3	UNI//BIP	1=unipolar, 0=bipolar
2	SGL//DIF	1= massebezogen, 0=Differenzeingänge
1	PD1	00: Power-Down 2µA, 01 30µA, schnell
0	PD0	10: Clock intern, 11 Clock extern

Die Steuerbits SEL0 bis SEL2 wählen den aktiven Eingangskanal aus. Ihre drei Bits stellen einen Zahlenwert im Bereich 0 bis 7 dar. Es stehen entweder vier Differenz-

eingänge oder acht massebezogene Eingänge zur Verfügung. Bipolare Messungen sind nur für Differenzeingänge möglich, weil die Eingangsspannungen nur im Bereich 0 bis 5V liegen können. Wählt man Differenzeingänge, dann muss der negative Eingang während der gesamten Umsetzzeit eine stabile Spannung aufweisen, während der positive Eingang nur kurz abgetastet wird und schnelle Änderungen verträgt.

Tabelle 11.3: Wahl der Eingangskanäle für Differenz- und Masse-bezogene Eingänge

SEL0...2	Differenzeingänge	Einzeleingang
0	CH0 gegen CH1	CH0
1	CH2 gegen CH3	CH2
2	CH4 gegen CH5	CH4
3	CH6 gegen CH7	CH6
4	CH1 gegen CH0	CH1
5	CH3 gegen CH2	CH3
6	CH5 gegen CH4	CH5
7	CH7 gegen CH6	CH7

Nach der Übertragung des Steuerbytes beginnt die eigentliche Umsetzung mit einer Dauer von maximal 10µs. Dann können zwei Bytes ausgelesen werden. Das zuerst gelesene Highbyte enthält die acht höchstwertigen Bits. Das Lowbyte enthält vier Datenbits in den Bits 4 bis 7 und Nullen in den vier niederwertigen Bits.

Im folgenden wird zunächst ein fertiges Anwenderprogramm für den Wandler beschreiben, danach die entsprechenden Ansteuerroutinen.

11.2 Das Anwenderprogramm Serai8/12 USB

Das Programm Serai8/12 USB wurde von H.-J. Berndt geschrieben. Es steht in einer Reihe mit weiteren Serai-Programmen (Serielles Analog Interface) für spezielle Analog-Interfaces an der seriellen Schnittstelle. Das Programm liegt in ausführbarer Form auf der CD. Es kann ohne die Hardware in einem Simulationsmodus verwendet werden. Das Programm besitzt ähnliche Menüs wie das weiter oben vorgestellte Compact2000 vom gleichen Autor. Auch hier können gemessene Daten mit externen Programmen weiterverarbeitet werden.

Beim Start lädt das Programm das Binärfile EZAD.BIN in den Controller. Diese Firmware ermöglicht die Ansteuerung des AD-Wandlers und die Übertragung von Daten.

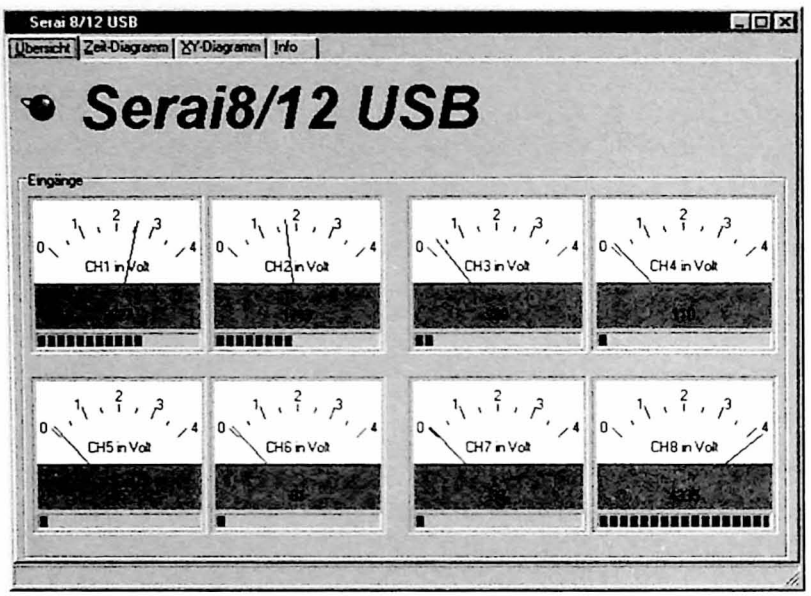


Abb. 11.3: Die Übersicht aller acht Kanäle

Die Software besitzt ein Menü zur Auswahl von drei Grundfunktionen. In der Übersicht werden alle acht Kanäle direkt angezeigt. Neben der Zeigerdarstellung erhält man eine einfache Balkenanzeige und eine Digitalanzeige. Entsprechend der Auflösung von 12 Bit hat die digitale Ausgabe einen Wertebereich von 0 bis 4095, zeigt also die Spannung in Millivolt.

In der Diagramm-Funktion werden zunächst alle acht Kanäle mit einstellbarer Geschwindigkeit gemeinsam gemessen. Der Anwender kann jedoch nachträglich alle Kanäle einzeln und in beliebiger Kombination betrachten. Außerdem lassen sich alle Daten mit einer speziellen Schaltfläche in die Zwischenablage kopieren, um sie z.B. in Excel weiter zu verarbeiten.

Gemessene Daten können im Tabellenformat mit einem Mausklick auf die entsprechende Schaltfläche in die Zwischenablage kopiert werden. Von dort lassen sie sich z.B. in Excel-Tabellen einfügen und weiter verarbeiten (vgl. *Abb. 11.5*).

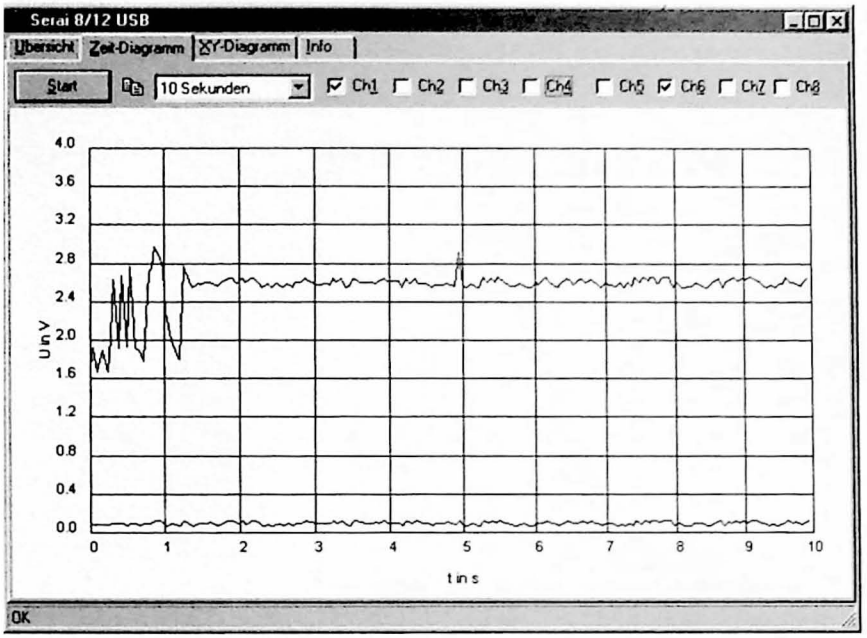


Abb. 11.4: Die Diagrammausgabe

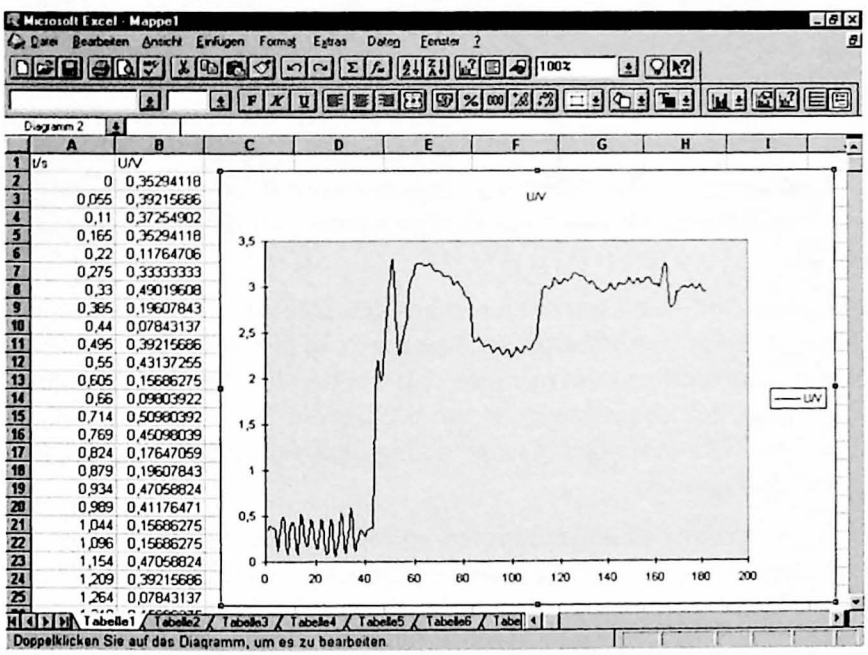


Abb. 11.5: Datenauswertung in Excel

Schließlich gibt es noch eine XY-Messung mit freier Auswahl der gegeneinander dargestellten Kanäle. Die Eingangskanäle und die Geschwindigkeit der Messung können über Schiebeschalter eingestellt werden.

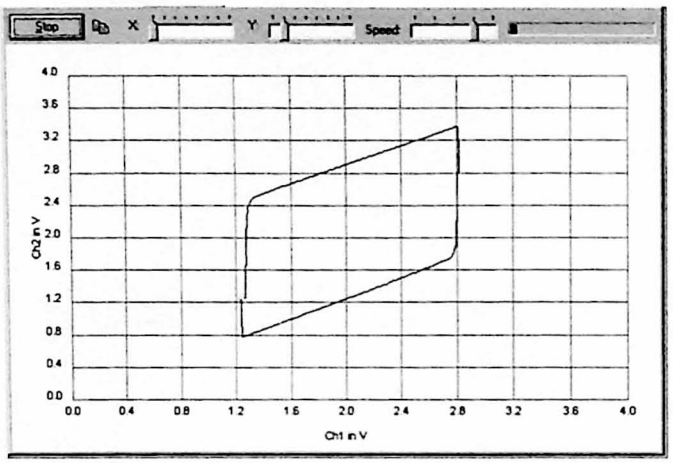


Abb. 11.6: Die XY-Darstellung

11.3 Assembler-Routine zum AD-Wandler

Das erste 8051-Ansteuerprogramm nach Listing 11.1 liest alle acht Eingangskanäle und schreibt die Ergebnisse in den Speicherbereich ab 200h ins RAM. Der PC kann hier über den USB insgesamt 16 Bytes auslesen. Zusätzlich wird der Port C ans Ausgangsport initialisiert. Vom Anwenderprogramm aus kann in die Adresse 210h ein Byte geschrieben werden, das vom Assemblerprogramm an das OUTC-Register übertragen wird.

Die eigentliche Wandlerroutine RdAD besteht aus drei Blöcken. Zunächst wird mit acht Taktimpulsen das im Akku übergebene Steuerbyte in den Wandler geschoben. Dann folgt eine kurze Wartezeit mit mehr als 10 µs, in der der Wandler die sukzessive Approximation des Eingangswertes durchführt. Auf eine Auswertung der SSTRB-Leitung wird hier verzichtet, da eine geringfügig verkürzte Wandlungszeit sich insgesamt kaum auswirkt.

Im zweiten Block werden acht Bits ausgelesen und in das Register R3 abgelegt. Im letzten Block werden mit vier Taktimpulsen die letzten vier Bits abgeholt und in R4 geschrieben.

Das übergeordnete Programm InOut fordert unipolare, massebezogene Messungen an allen acht Kanälen an und kopiert die Messergebenisse in den vorgesehenen

RAM-Bereich. Die jeweiligen Steuerbytes sind fest kodiert. Schließlich wird auch noch das Ausgangsbyte aus der RAM-Adresse 210h in das OUTC-Register kopiert.

Das gesamte Programm läuft in einer Schleife endlos ab. Die Datenübertragung erfolgt völlig asynchron zur Messung allein durch den USB-Kern. Die Host-Software führt dazu Control-Zugriffe über Endpoint 0 durch.

```
;Max186a.asm
#include 8051.h
OEA  .equ   7F9Ch
OEB  .equ   7F9Dh
OEC  .equ   7F9Eh
PINSA .equ  7F99h
PINSB .equ  7F9Ah
PINSC .equ  7F9Bh
OUTA .equ   7F96h
OUTB .equ   7F97h
OUTC .equ   7F98h
DPS  .equ   86h

     lcall initMAX186
     lcall initPortC
Loop lcall InOut
     sjmp Loop

initPortC
     mov DPTR,#OUTC
     mov A,#255
     movx @DPTR,A
     ret

initMAX186
     mov DPTR,#OEB
     mov a,#7                    ;PortB0...2 Outputs
     movx @DPTR,a
     mov DPTR,#OUTB
     mov a,#2                    ;/CS = 1
     movx @DPTR,a
     ret

Delay                            ;delay [a] times
     mov R1,#255
S0   djnz R1,S0
     dec a
     jnz Delay
     ret

RdAD                             ;Control byte in A
     mov R3,a                    ;save Control Byte
```

```
     mov DPTR,#OUTB
     mov a,0                    ;/CS = 0
     movx @DPTR,a
;*** Transfer control byte ***
     mov A,#0
     mov R2,#8
S1   Xch a,R3                   ;Control-Byte
     jnb ACC.7,S2               ;Bit7 = 1?
     mov R3,#4                  ;Din = 1
S2   xch a,R3
     movx @DPTR,a
     inc a                      ;SCLK = 1
     movx @DPTR,a
     dec a
     movx @DPTR,a               ;SCLK = 0
     mov a,#0                   ;Din =0
     xch a,R3
     rl  a                      ;next Bit
     xch a,R3
     djnz R2,S1
;*** Wait for conversion complete ***
     mov R2,#30
S3   djnz R2,S3                 ;delay > 10 µs
;*** Read highbyte to R3 ***
     mov R3,#0                  ;Highbyte > R3
     mov R2,#8
S4   xch a,R3
     rl  a
     xch a,R3
     mov DPTR,#OUTB
     mov a,#1                   ;SCLK = 1
     movx @DPTR,a
     mov a,#0                   ;SCLK = 0
     movx @DPTR,a
     mov DPTR,#PINSB
     movx a,@DPTR
     jnb ACC.4,S5
     inc R3
S5   djnz R2,S4
;*** Read lowbyte to R4 ***
     mov R4,#0                  ;Lowbyte > R4
     mov R2,#4
S6   xch a,R4
     rl  a
     xch a,R4
     mov DPTR,#OUTB
     mov a,#1                   ;SCLK = 1
     movx @DPTR,a
     mov a,#0                   ;SCLK = 0
```

```
     movx @DPTR,a
     mov DPTR,#PINSB
     movx a,@DPTR
     jnb ACC.4,S7
     inc R4
S7   djnz R2,S6

     mov DPTR,#OUTB
     mov a,#2                    ;/CS = 1
     movx @DPTR,a
     ret

InOut
     mov a,#142                  ;Ch0
     lcall RdAD
     mov a,R3                    ;Highbyte
     mov DPTR,#0200h
     movx @DPTR,A
     mov a,R4                    ;Lowbyte
     inc DPTR
     movx @DPTR,A

     mov a,#206                  ;Ch1
     lcall RdAD
     mov a,R3                    ;Highbyte
     mov DPTR,#0202h
     movx @DPTR,A
     mov a,R4                    ;Lowbyte
     inc DPTR
     movx @DPTR,A

     mov a,#158                  ;Ch2
     lcall RdAD
     mov a,R3                    ;Highbyte
     mov DPTR,#0204h
     movx @DPTR,A
     mov a,R4                    ;Lowbyte
     inc DPTR
     movx @DPTR,A

     mov a,#222                  ;Ch3
     lcall RdAD
     mov a,R3                    ;Highbyte
     mov DPTR,#0206h
     movx @DPTR,A
     mov a,R4                    ;Lowbyte
     inc DPTR
     movx @DPTR,A
```

```
    mov a,#174              ;Ch4
    lcall RdAD
    mov a,R3                ;Highbyte
    mov DPTR,#0208h
    movx @DPTR,A
    mov a,R4                ;Lowbyte
    inc DPTR
    movx @DPTR,A

    mov a,#238              ;Ch5
    lcall RdAD
    mov a,R3                ;Highbyte
    mov DPTR,#020Ah
    movx @DPTR,A
    mov a,R4                ;Lowbyte
    inc DPTR
    movx @DPTR,A

    mov a,#190              ;Ch6
    lcall RdAD
    mov a,R3                ;Highbyte
    mov DPTR,#020Ch
    movx @DPTR,A
    mov a,R4                ;Lowbyte
    inc DPTR
    movx @DPTR,A

    mov a,#254              ;Ch7
    lcall RdAD
    mov a,R3                ;Highbyte
    mov DPTR,#020Eh
    movx @DPTR,A
    mov a,R4                ;Lowbyte
    inc DPTR
    movx @DPTR,A

    mov DPTR,#0210h
    movx a,@DPTR
    mov DPTR,#OUTC
    movx @DPTR,a

    mov a,#1
    lcall Delay
    ret

.End
```

Listing 11.1: Ansteuerung des MAX186

Das Programm benötigt ca. 2 ms, um alle 8 Kanäle zu lesen und das Ausgangsbyte zu übertragen.

Das Delphi-Programm nach Listing 11.2 liest alle acht Eingangskanäle und zeigt die Rohdaten in Form von Zahlenwerten zwischen 0 und 4095 am Bildschirm an. Die Schaltfläche „Test Port C Output" dient zum Test der Portausgabe. Aus der Unit EZUSB.PAS wird die Funktion ReadRAMBytes verwendet, um alle Wandler-Daten in einem Block zu lesen.

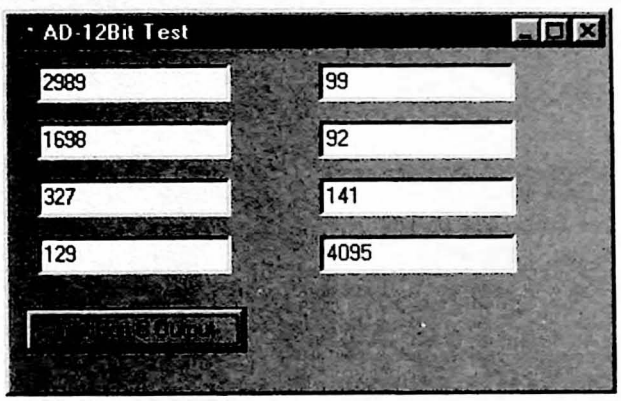


Abb. 11.7: Auslesen aller acht Analog-Kanäle

```
unit Max186a;

interface

uses EZUSB,
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Test: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Edit4: TEdit;
    Edit5: TEdit;
    Edit6: TEdit;
    Edit7: TEdit;
    Edit8: TEdit;
    Timer1: TTimer;
    procedure TestClick(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
    procedure FormCreate(Sender: TObject);
```

```
  private
    { Private-Deklarationen}
  public
    { Public-Deklarationen}
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.TestClick(Sender: TObject);
var InBuffer: Array [1..16] of Byte;
    Messwert: Word;
begin
  WrRAM ($0210,85);    //Output Port C
end;

procedure TForm1.Timer1Timer(Sender: TObject);
var     Messwert: Word;
begin
  ReadRamBytes (512,16);
  Messwert := InBuffer[1]+16*InBuffer[0];
  Edit1.Text := FloatToStr (Messwert);
  Messwert := InBuffer[3]+16*InBuffer[2];
  Edit2.Text := FloatToStr (Messwert);
  Messwert := InBuffer[5]+16*InBuffer[4];
  Edit3.Text := FloatToStr (Messwert);
  Messwert := InBuffer[7]+16*InBuffer[6];
  Edit4.Text := FloatToStr (Messwert);
  Messwert := InBuffer[9]+16*InBuffer[8];
  Edit5.Text := FloatToStr (Messwert);
  Messwert := InBuffer[11]+16*InBuffer[10];
  Edit6.Text := FloatToStr (Messwert);
  Messwert := InBuffer[13]+16*InBuffer[12];
  Edit7.Text := FloatToStr (Messwert);
  Messwert := InBuffer[15]+16*InBuffer[14];
  Edit8.Text := FloatToStr (Messwert);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 ProgReset;
 sleep (10);
 DownLoadIntelHex ('MAX186_a.obj');
 sleep(10);
 ProgStart;
end;
end.
```

Listing 11.2: Anzeige der Messdaten

11.4 Ein Speicher-Oszilloskop

Die hohe Übertragungsrate über den USB ermöglicht den Bau komfortabler Speicher-Oszilloskope. Hier wird zunächst ein Einkanal-Programm mit einer Abtastrate von 10000 Messungen pro Sekunde beschrieben.

Die 8051-Steuersoftware nach Listing 11.3 führt in schneller Folge 1024 Messungen an Kanal 0 durch. Aus Zeitgründen werden nach jeder Messung nur acht Bits ausgelesen. Damit kommt man auf die gewünschte Abtastrate von 10 kHz. Die geringere Auflösung von 8 Bit entspricht etwa der Bildschirmauflösung.

```
;Max186b.asm
;Einkanal-Oszilloskop, nur 8 Bit, 10 kS/s

#include 8051.h
OEA  .equ   7F9Ch
OEB  .equ   7F9Dh
OEC  .equ   7F9Eh
PINSA .equ  7F99h
PINSB .equ  7F9Ah
PINSC .equ  7F9Bh
OUTA .equ   7F96h
OUTB .equ   7F97h
OUTC .equ   7F98h
DPS  .equ   86h

     lcall initMAX186
   ;     lcall initPortC
     mov DPTR,#0200h
     mov r6,#30
L1   mov r7,#0
L2   mov a,#142   ;Ch0
     inc DPS
     lcall RdAD
     inc DPS
     mov a,R3  ;Highbyte
     movx @DPTR,A
     inc DPTR
     ;mov a,R4 ;Lowbyte
     ;movx @DPTR,A
     ;inc DPTR
     djnz r7,L2
     djnz r6,L1
Loop sjmp Loop

initPortC
     mov DPTR,#OUTC
     mov A,#255
```

```
     movx @DPTR,A
     ret

initMAX186
     mov DPTR,#0EB
     mov a,#7     ;PortB0...2 Outputs
     movx @DPTR,a
     mov DPTR,#0UTB
     mov a,#2     ;/CS = 1
     movx @DPTR,a
     ret

Delay     ;delay [a] times
     mov R1,#255
S0   djnz R1,S0
     dec a
     jnz Delay
     ret

RdAD      ;Control byte in A
     mov R3,a  ;save Control Byte
     mov DPTR,#0UTB
     mov a,0   ;/CS = 0
     movx @DPTR,a
;*** Transfer control byte ***
     mov A,#0
     mov R2,#8
S1   Xch a,R3  ;Control-Byte
     jnb ACC.7,S2 ;Bit7 = 1?
     mov R3,#4    ;Din = 1
S2   xch a,R3
     movx @DPTR,a
     inc a     ;SCLK = 1
     movx @DPTR,a
     dec a
     movx @DPTR,a ;SCLK = 0
     mov a,#0  ;Din =0
     xch a,R3
     rl  a     ;next Bit
     xch a,R3
     djnz R2,S1
;*** Wait for conversion complete ***
     mov R2,#30
S3   djnz R2,S3   ;delay > 10 µs
;*** Read highbyte to R3 ***
     mov R3,#0 ;Highbyte > R3
     mov R2,#8
S4   xch a,R3
     rl  a
```

```
      xch a,R3
      mov DPTR,#OUTB
      mov a,#1      ;SCLK = 1
      movx @DPTR,a
      mov a,#0      ;SCLK = 0
      movx @DPTR,a
      mov DPTR,#PINSB
      movx a,@DPTR
      jnb ACC.4,S5
      inc R3
S5    djnz R2,S4
      mov DPTR,#OUTB
      mov a,#2      ;/CS = 1
      movx @DPTR,a
      ret

.End
```

Listing 11.3: Eine schnelle Serienmessung

Das Delphi-Programm USB-Scope stellt die Messwerte auf einem Schirm mit 255 mal 500 Pixeln dar. Entsprechend werden nur 500 Messpunkte aus dem Controller-RAM gelesen. Alle Messungen werden per Mausklick über die Start-Schaltfläche ausgelöst.

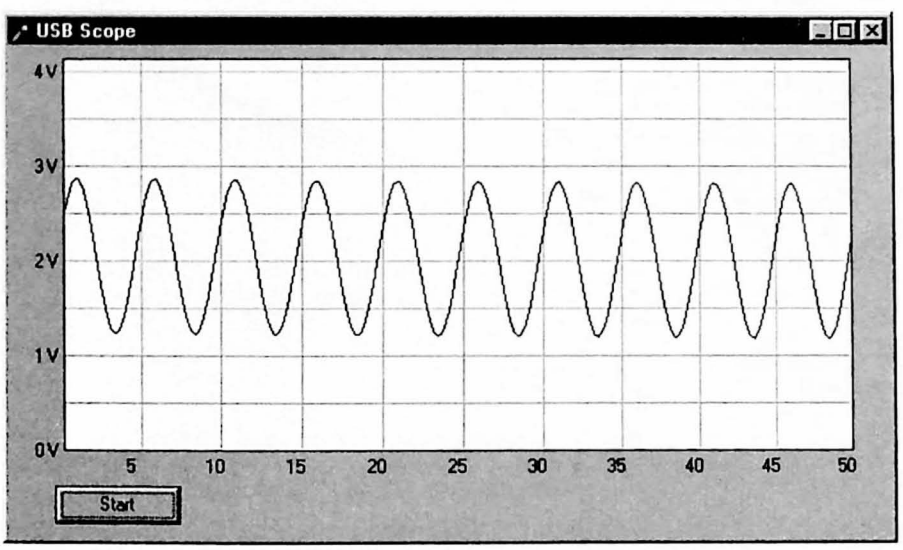


Abb. 11.8: Messung an einem 2-kHz-Sinussignal

```
unit Oszi;

interface

uses EZUSB,
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    PaintBox1: TPaintBox;
    Messung: TButton;
    procedure MessungClick(Sender: TObject);
    procedure PaintBox1Click(Sender: TObject);
  end;

var
  Form1: TForm1;
  Buffer : Array [0..8000] of Byte;

implementation

{$R *.DFM}

procedure SpeichernTabelle (Dateiname :String);
var  f : Text;
     n, r : Integer;
begin
  AssignFile(f,dateiname);
  {$I-} ReWrite(f); {$I+}
  r := IOresult;
  if  r = 0 then begin
     for n:=0 to 999 do begin;
       write (f,Buffer [4*n+1],#9);
       write (f,Buffer [4*n+2],#9);
       write (f,Buffer [4*n+3],#9);
       writeln (f,Buffer [4*n+4]);
     end;
     CloseFile (f);
  end;
end;

procedure DiagrammInit;
var n: Integer;
begin
  with Form1.Paintbox1.Canvas do begin
  Pen.Color:=ClBlack;
  Brush.Color:=ClWhite;
  Rectangle(30,0,531,257);
  Pen.Color:=ClLtGray;
  for n:= 1 to 8 do begin
      MoveTo(31,256-n*31);
      Lineto (529,256-n*31);
```

```
  end;
  Brush.Color:=ClLtGray;
  for n:= 0 to 4 do begin
    TextOut(10,250-n*62,FloatToStr(n)+' V');
  end;
  for n:= 1 to 9 do begin
      MoveTo(30+50*n,254);
      Lineto (30+50*n,0)
  end;
  for n:= 1 to 10 do begin
    TextOut(20+50*n,260,FloatToStr(n*5));
  end;
  end;
end;

procedure Plot (Kanal: Integer);
var i: Integer;
begin
  with Form1.Paintbox1.Canvas do begin
    Pen.Color:=ClBlack;
    MoveTo (31,256-Buffer[0]);
    for i := 1 to 499 do
      LineTo (31+i ,256-Buffer[i]);
  end;
end;

procedure TForm1.MessungClick(Sender: TObject);
var n,i: Word;
begin
  DiagrammInit;
  ProgReset;
  DownLoadBin ('MAX186_b.obj');
  ProgStart;
  Sleep (100);
  ProgReset;
  for n:= 0 to 3 do begin
    ReadRamBytes ((512+n*1024),1024);
    for i:= 0 to 1023 do Buffer[n*1024+i] := InBuffer[i];
  end;
  Plot (0)
end;

procedure TForm1.PaintBox1Click(Sender: TObject);
begin
DiagrammInit;
end;

end.
```

Listing 11.4: Das Oszilloskop in Delphi

11.5 Triggerung

Jedes moderne Oszilloskop hat eine Triggerfunktion, mit der der Anfangspunkt eines Oszillogramms relativ zum Signal festgelegt werden kann. Die Messung beginnt erst, wenn die Eingangsspannung einen bestimmten Triggerlevel überschritten (positive Flanke) oder unterschritten (negative Flanke) hat. Bei einem analogen Oszilloskop ermöglicht diese Funktion stehende Bilder bei periodischen Signalen. Digitale Oszilloskope ermöglichen zwar durch Einzeldurchgänge immer stehende Bilder. Oft ist es aber schwierig, ein bestimmtes Ereignis aufzuzeichnen. Hier hilft eine Triggerfunktion.

Das Assemblerprogramm aus Kap. 11.3 kann mit einer kleinen Änderung mit einer Triggerfunktion ausgestattet werden. Hier wird immer mit positiver Triggerschwelle bei der halben Eingangsspannung gearbeitet. Ein Programmabschnitt vor der eigentlichen Messung wartet auf einen Spannungsdurchgang durch den Triggerpunkt.

Zuvor muss noch der Messwertspeicher gelöscht werden. Da nämlich das Auslesen über den USB völlig unabhängig von der eigentlichen Messung erfolgt, kann der Anwender nicht ohne weiteres erkennen, ob das Triggerereignis bereits eingetreten ist. Er würde die Speicherinhalte der letzen Messung erhalten. Hier wird aber der gesamte Speicher mit dem Wert 1 gefüllt, also quasi gelöscht. Der Anwender sieht eine gerade Linie am unteren Rand des Schirms, wenn die Messung nicht ausgelöst wurde.

```
;Max186c.asm
;Einkanal-Oszilloskop, nur 8 Bit, 10 kS/s mit Triggern

#include 8051.h
OEA  .equ   7F9Ch
OEB  .equ   7F9Dh
OEC  .equ   7F9Eh
PINSA .equ  7F99h
PINSB .equ  7F9Ah
PINSC .equ  7F9Bh
OUTA .equ   7F96h
OUTB .equ   7F97h
OUTC .equ   7F98h
DPS  .equ   86h

     lcall initMAX186
   ;     lcall initPortC

     mov DPTR,#0200h
     mov r6,#4
C1   mov r7,#0
```

```
C2   mov a,#1      ;clear/fill RAM
     movx @DPTR,A
     inc DPTR
     djnz r7,C2
     djnz r6,C1

     mov r7,#127
Tr1  mov a,#142   ;Ch0
     lcall RdAD
     mov a,R3
     subb a,R7
     jnc  Tr1
Tr2  mov a,#142   ;Ch0
     lcall RdAD
     mov a,R3
     subb a,R7
     jc  Tr2

     ...
```

Listing 11.5: Triggern mit konstanter Schwelle bei 127

Die eigentliche Triggerfunktion wird in zwei Abfrageschleifen gebildet. Die Schwelle befindet sich im Register R7. Die erste Schleife wird so lange durchlaufen, bis der Messwert unter der Schwelle liegt. Dies wird in vielen Fällen bereits vor dem Start des Programms gegeben sein, verhindert jedoch, dass die Messung beginnt, wenn die Eingangsspannung gerade höher liegt. Beim Eintritt in die zweite Schleife ist also gewährleistet, dass die Eingangsspannung tiefer liegt. In der zweiten Schleife wird dann so lange gewartet, bis die Eingangsspannung die Triggerschwelle überschreitet. Danach beginnt unmittelbar die eigentliche Serienmessung.

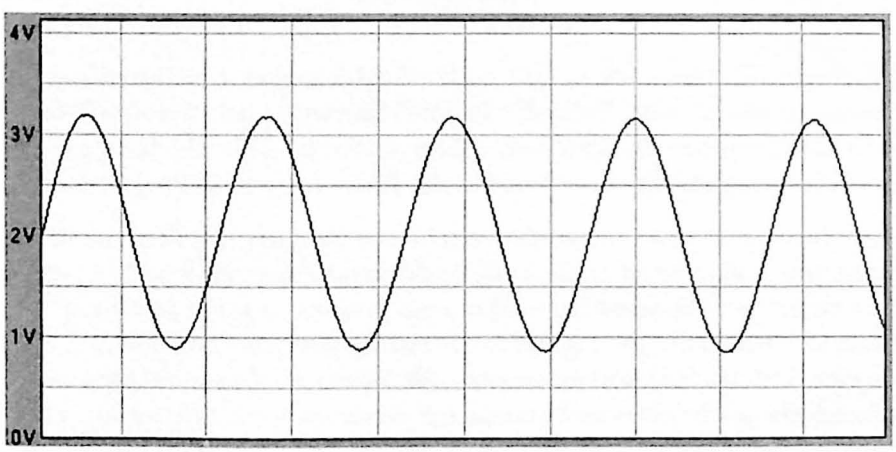


Abb. 11.9: Messung, getriggert bei Level 127

Das assemblierte Programm kann direkt mit dem einfachen Oszilloskop OSZI.PAS verwendet werden. *Abb. 11.9* zeigt ein Messergebnis.

Für den praktischen Einsatz des Geräts ist es wichtig, die Triggerschwelle frei wählen zu können. Sie soll deshalb in einer Speicherstelle im RAM des Prozessors übergeben werden. Im Programm MAX186d.asm wurde die Adresse 1FFh vor dem eigentlichen Datenbereich ab 200h gewählt.

```
mov DPTR,#0200h
     mov r6,#4
C1   mov r7,#0
C2   mov a,#1          ;clear/fill RAM
     movx @DPTR,A
     inc DPTR
     djnz r7,C2
     djnz r6,C1

     mov DPTR,#1FFh
     movx a,@DPTR
     mov r7,a          ;#127
Tr1  mov a,#142        ;Ch0
     lcall RdAD
     mov a,R3
     subb a,R7
     jnc  Tr1
Tr2  mov a,#142        ;Ch0
     lcall RdAD
     mov a,R3
     subb a,R7
     jc  Tr2
```

Listing 11.6: Triggerung mit wählbarem Triggerlevel

Das Delphi-Programm Oszi2 wurde mit einem Schieberegler zur Einstellung der Triggerschwelle versehen. In der Prozedur FormCreate wird zunächst eine Default-Schwelle von 127 übergeben. In ScrollBar1.Change wird bei jeder Änderung an der Position des Schiebereglers ein neuer Wert in die RAM-Adresse 1FFh geschrieben.

Einige weitere Bedienelemente dienen dem leichteren Umgang mit dem Gerät. Außer der manuellen Auslösung einer Messung wurde eine Timer-gesteuerte Automatik-Funktion eingebaut. Die Messung wird einmal pro Sekunde wiederholt, so dass der Anwender sich besser auf die eigentliche Messung konzentrieren kann. Außerdem wurde eine Speicherfunktion eingebaut. Die Daten werden im Textformat in eine Ausgabetabelle geschrieben und stehen zur weiteren Verarbeitung mit externen Programmen bereit.

```
unit Oszi2;

interface

uses EZUSB,
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls;

...

procedure SaveData (Dateiname :String);
var  f : Text;
     n, r : Integer;
begin
  AssignFile(f,dateiname);
  {$I-} ReWrite(f); {$I+}
  r := IOresult;
  if  r = 0 then begin
     for n:=0 to 1023 do begin;
        writeln (f,Buffer [n]);
     end;
     CloseFile (f);
  end;
end;

procedure TForm1.SaveClick(Sender: TObject);
begin
  SaveData('Output.dat');
end;

procedure TForm1.MessungClick(Sender: TObject);
var n,i: Word;
begin
  ProgReset;
  DiagrammInit;
  ProgStart;
  Sleep (100);
  ProgReset;
  for n:= 0 to 3 do begin
    ReadRamBytes ((512+n*1024),1024);
    for i:= 0 to 1023 do Buffer[n*1024+i] := InBuffer[i];
  end;
  Plot (0)
end;
...

procedure TForm1.ScrollBar1Change(Sender: TObject);
begin
  WrRAM ($01FF,255-Scrollbar1.Position);
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  ProgReset;
  DownLoadBin ('MAX186_d.obj');
```

```
  WrRAM ($01FF,127); //Trigger = 127
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Timer1.Enabled := true;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
var n,i: Word;
begin
  ProgReset;
  DiagrammInit;
  ProgStart;
  Sleep (200);
  ProgReset;
  for n:= 0 to 3 do begin
    ReadRamBytes ((512+n*1024),1024);
    for i:= 0 to 1023 do Buffer[n*1024+i] := InBuffer[i];
  end;
  Plot (0)
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Timer1.Enabled := false;
end;

end.
```

Listing 11.7: Das erweiterte Oszilloskop

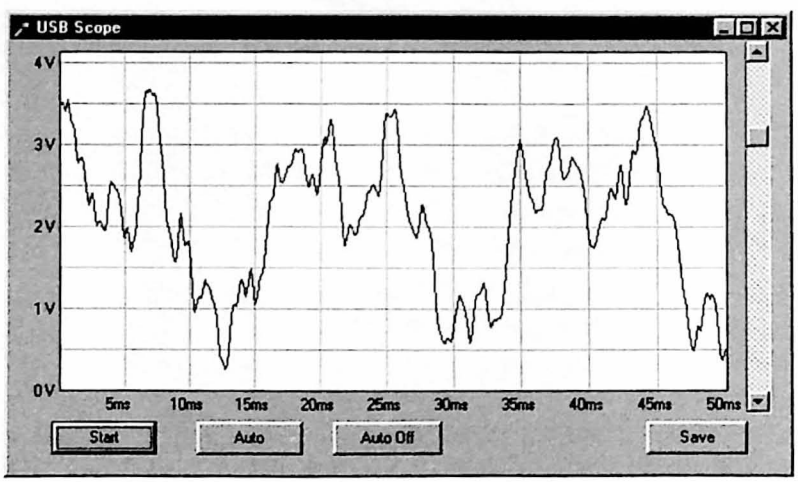


Abb. 11.10: Messergebnis an einem Audiosignal

Die gewonnenen Messdaten können mit der SAVE-Schaltfläche in die Datei Output.Dat übertragen werden. Diese Daten können z.B. in Excel eingelesen und verarbeitet werden. *Abb. 11.11* zeigt die Darstellung eines Teilbereichs der gemessenen Daten.

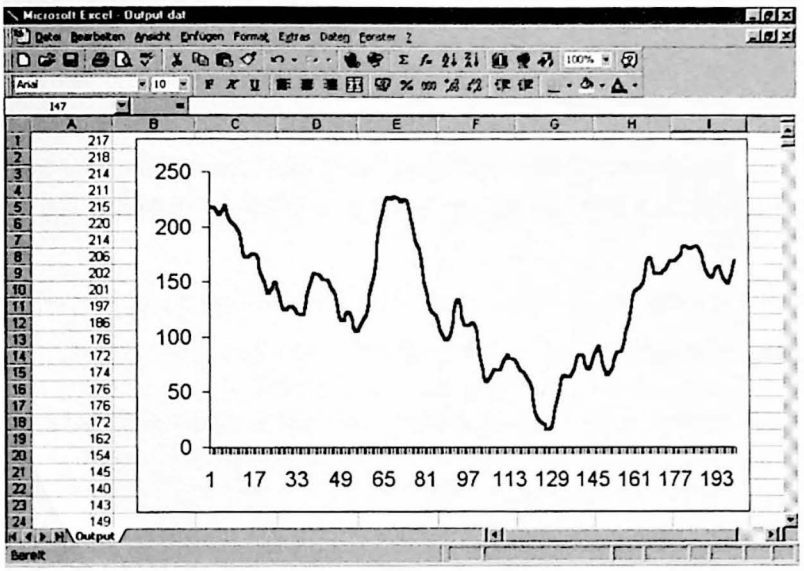


Abb. 11.11: Verarbeitung der Daten in Excel ((Scope4.gif))

12 Der I²C-Bus

Der I²C-Bus (Inter-IC-Bus) ist eine 2-Draht-Datenverbindung zwischen einem oder mehreren Prozessoren (Mastern) und speziellen Peripheriebausteinen (Slaves). Alle Bausteine liegen am selben Bus und werden gezielt unter ihren Adressen angesprochen. Adressen und Daten werden über dieselben Leitungen übertragen. Der Bus erlaubt eine sehr einfache Verbindung zwischen vielen ICs und problemlose nachträgliche Erweiterungen.

12.1 Das Busprotokoll

Es können alle ICs angeschlossen werden, die das spezielle Busprotokoll beherrschen. Neben RAMs, EEPROMs, Porterweiterungsbausteinen, A/D- und D/A-Wandlern und Uhrenbausteinen gibt es eine ganze Reihe von speziellen ICs wie z.B. Anzeigentreiber oder ICs für die Fernsehtechnik.

Der I²C-Bus verwendet die serielle Datenleitung SDA und die Taktleitung SCL. Die Daten und Adressen werden wie bei Schieberegistern zusammen mit einem Takt übertragen. Beide Leitungen können in jeder Datenrichtung verwendet werden. Sie sind jeweils mit einem Pullup-Widerstand versehen und können von jedem Teilnehmer durch Open-Collector- oder Open-Drain- Ausgänge auf Low-Pegel gezogen werden. *Abb. 12.1* zeigt das Prinzip der Busverbindung. Inaktive Busteilnehmer sind hochohmig, werten jedoch immer die Bus-Signale aus. Wenn nur ein Master verwendet wird, gibt dieser allein das Taktsignal aus. Die Daten können jedoch sowohl vom Master als auch vom Slave kommen.

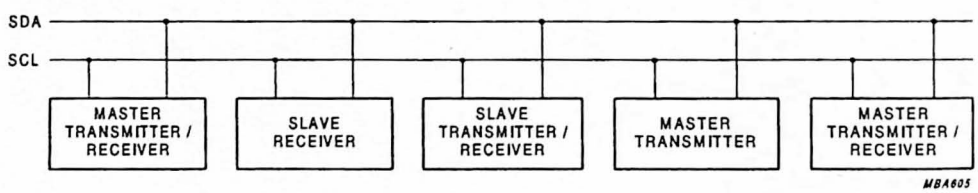


Abb. 12.1: I²C-Bus-Verbindungen zwischen Master- und Slave-ICs (Quelle: Philips)

Der EZ-USB-Chip verfügt über eine interne Hardware zur Ansteuerung des I²C-Bus. Die beiden Leitungen SCL und SDA stehen dazu unabhängig von den Portleitungen bereit. Sie dienen einerseits für die allgemeine Ansteuerung von Peripherie. Andererseits kann hier ein I²C-EEPROM angeschlossen werden, über das beim Start des Systems Programme oder Deskriptoren geladen werden können.

Das I²C-Bus-Protokoll kennt eine Reihe genau definierter Situationen, die es jedem Busteilnehmer gestatten, Anfang und Ende einer Übertragung sowie seine mögliche Adressierung zu erkennen:

- Ruhezustand: SDA und SCL sind high und damit inaktiv.
- Startbedingung: SDA wird vom Master heruntergezogen, während SCL high bleibt.
- Stopbedingung: SDA wechselt von low nach high, während SCL high bleibt.
- Datenübertragung: Der jeweilige Sender legt acht Datenbits auf die Datenleitung SDA, die durch Taktimpulse auf der Clockleitung SCL vom Master weitergeschoben werden. Die Übertragung beginnt mit dem höchstwertigen Bit.
- Bestätigung (Acknowledge): Der jeweilige Empfänger quittiert den Empfang eines Bytes durch einen Low-Pegel an SDA, bis der Master den neunten Clock-Impuls an SCL erzeugt hat. Die Bestätigung bedeutet zugleich, dass ein weiteres Byte empfangen werden soll. Ein gewünschtes Ende der Übertragung muss durch das Ausbleiben der Bestätigung angekündigt werden. Die eigentliche Beendigung der Übertragung wird durch die Stopbedingung erreicht.

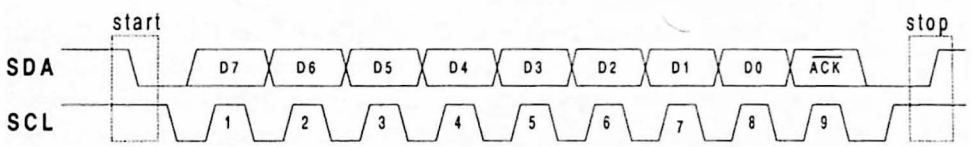


Abb. 12.2: Datenübertragung über den I²C-Bus

Adressen werden genau wie Daten übertragen und quittiert. Im einfachsten Fall einer Datenübertragung vom Master zu einem Slave, z.B. einem Ausgabeport, laufen folgende Vorgänge ab: Der Master erzeugt die Startbedingung und überträgt dann in den Bits 7 bis 1 die Adresse des Portbausteins und in Bit 0 die gewünschte Richtung der Datenübertragung, nämlich 0 für „Schreiben". Die Adresse wird vom angesprochenen Slave quittiert. Dann sendet der Master das Datenbyte, das ebenfalls quittiert wird. Er kann nun die Verbindung durch die Stopbedingung unterbrechen oder aber weitere Bytes an denselben Slave abschicken.

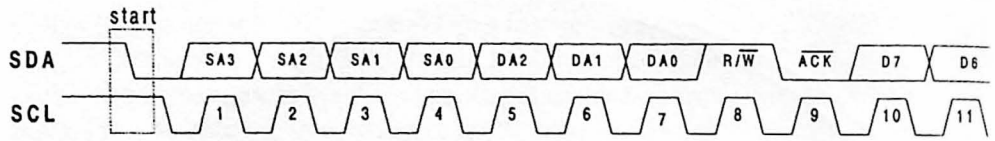


Abb. 12.3: Übertragung einer Adresse

Sollen Daten von einem Slave gelesen werden, muss die Adresse mit gesetztem Datenrichtungsbit R/W übertragen werden. Der Master gibt jeweils acht Clock-Impulse aus und erhält acht Datenbits. Solange er den Empfang beim neunten Clock-Impuls durch Acknowledge bestätigt, kann er weitere Bytes empfangen. Die Übertragung wird schließlich vom Master durch eine ausbleibende Bestätigung und die Stopbedingung beendet.

Jeder I²C-Baustein hat eine festgelegte Adresse, die zu einem Teil typenspezifisch (SA0...SA3) festgelegt ist und zu einem anderen Teil über herausgeführte Adressleitungen (DA0..DA2) veränderlich ist. Bei drei herausgeführten Adressleitungen können bis zu acht Bausteine des gleichen Typs am Bus liegen.

Die maximale Taktrate für den I²C-Bus beträgt für die meisten ICs 100 kHz. Der EZ-USB-Chip verwendet bei einer Quarzfrequenz von 12 MHz eine Taktrate von 90,9 kHz.

12.2 Steuerregister

Die interne I²C-Einheit des AN2131 besitzt zwei Register. Das I²C-Datenregister I2DAT (Adresse 7FA6) nimmt die acht Datenbits einer Schreib- oder Leseaktion auf. Über das I²C-Control- und Statusregister I2CS (Adresse 7FA5) werden die einzelnen Funktionen gesteuert. Die Bits 5...7 werden vom Anwenderprogramm gesetzt, die Bits 0...4 sind Statusbits und können nur gelesen werden.

Tabelle 12.1: Das I2CS-Register an der Adresse 7FA5h

Bit	Name	Funktion
7	START	Start = 1 sendet beim nächsten Schreibzugriff auf I2DAT eine Startbedingung vor den Daten. Das Bit wird automatisch gelöscht.
6	STOP	Stop = 1 sendet die Stop-Bedingung. Das Bit wird automatisch gelöscht.

Bit	Name	Funktion
5	LASTRD	Lastread = 1 wird vor dem letzten Lesezugriff auf I2DAT gesetzt, um ACK zu unterdrücken. Das Bit wird automatisch gelöscht.
4	ID1	ID1 und ID0 werden durch den Boot-Loader gesetzt:
3	ID0	00= Kein EEPROM, 01 Ein-Byte-Adresse erkannt 10: Zwei-Byte-Adresse erkannt
2	BERR	BERR=1 zeigt einen Bus-Fehler an
1	ACK	ACK=1 zeigt Acknowlegde des Slaves an
0	DONE	DONE = 1 zeigt das Ende einer Byte-Übertragung

Jeder vollständig abgeschlossene I²C-Transfer wird durch das DONE-Bit angezeigt. Durch Polling dieses Bits kann ein Programm die Übertragung abwarten. Daneben steht auch noch ein Interrupt zur Verfügung. Wenn der I²C-Interrupt freigegeben wurde, wird gleichzeitig mit dem Setzen des DONE-Bits auch der Int3 ausgelöst. In den folgenden Beispielen wird jedoch mit Polling gearbeitet.

Um mehrere Bytes über den I²C-Bus zu senden, sind die folgenden Schritte erforderlich:

1. START-Bit in I2CS setzen
2. Die I²C-Adresse mit R/W=0 in I2DAT schreiben
3. Warten, bis DONE in I2CS gesetzt ist
4. Das erste Datenbyte in I2DAT schreiben
5. Warten, bis DONE in I2CS gesetzt ist
6. Schritt 4 und 5 für jedes zu sendende Byte wiederholen
7. STOP in I2CS setzen

Zusätzlich zur Abfrage des DONE-Bits kann auch das ACK-Bit abgefragt werden, um festzustellen, ob das angesprochene IC am Bus antwortet. Das BERR-Bit kann außerdem beobachtet werden, um andere Fehler zu erkennen.

Um eines oder mehrere Byte von einem I²C-Slave zu lesen, sind folgende Schritte erforderlich:

1. START-Bit in I2CS setzen
2. Die I²C-Adresse mit R/W=0 in I2DAT schreiben
3. Warten, bis DONE in I2CS gesetzt ist

4. Lesen eines Dummy-Bytes von I2DAT, um die Übertragung zu starten
5. Warten, bis DONE in I2CS gesetzt ist
6. Lesen des ersten Datenbytes von I2DAT
7. Wiederholen von Schritt 5 und 6 bis zum vorletzten zu lesenden Byte
8. LASTRD in I2CS setzen
9. Lesen des letzen Bytes aus I2DAT
10. STOP in I2CS setzen
11. Ein Dummybyte aus I2DAT lesen

12.3 Portexpander PCF8574

Der PCF8574 ist ein 8-Bit-Portexpander mit I²C-Bus. Die Busadresse ist 0100xxx0 beim Schreiben und 0100xxx1 beim Lesen, wobei die drei veränderbaren Bits xxx durch die externen Adresseingänge A0 bis A2 festgelegt werden.

Der Baustein ist für Betriebsspannungen zwischen 2,5 V und 6 V ausgelegt. Er kann damit direkt an das EZ-USB-Board mit 3,3 V angeschlossen werden.

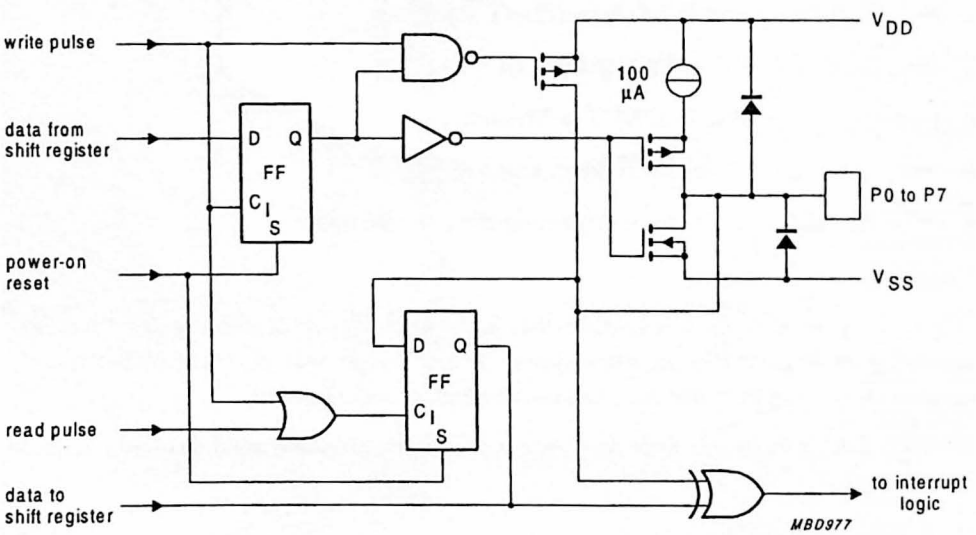


Abb. 12.4: Das Blockschaltbild des Portexpanders PCF8574 (Philips)

Das IC verfügt über acht quasi-bidirektionale Ports, wie sie auch beim Standard-8051 verwendet werden. Eine als Ausgang hochgesetzte Leitung verfügt über interne Pullup-Widerstände und ist hochohmig. Sie kann von außen auf Massepegel gezogen werden. Eine durch einen Ausgabezugriff heruntergezogene Leitung ist dagegen niederohmig. Nach einem Reset durch Anlegen der Betriebsspannung sind alle Leitungen hochgesetzt und können als Eingänge verwendet werden. Will man an den Ausgängen ohne einen zusätzlichen Treiber LEDs anschließen, dann müssen sie mit invertierter Logik gegen die positive Betriebsspannung angeschlossen werden.

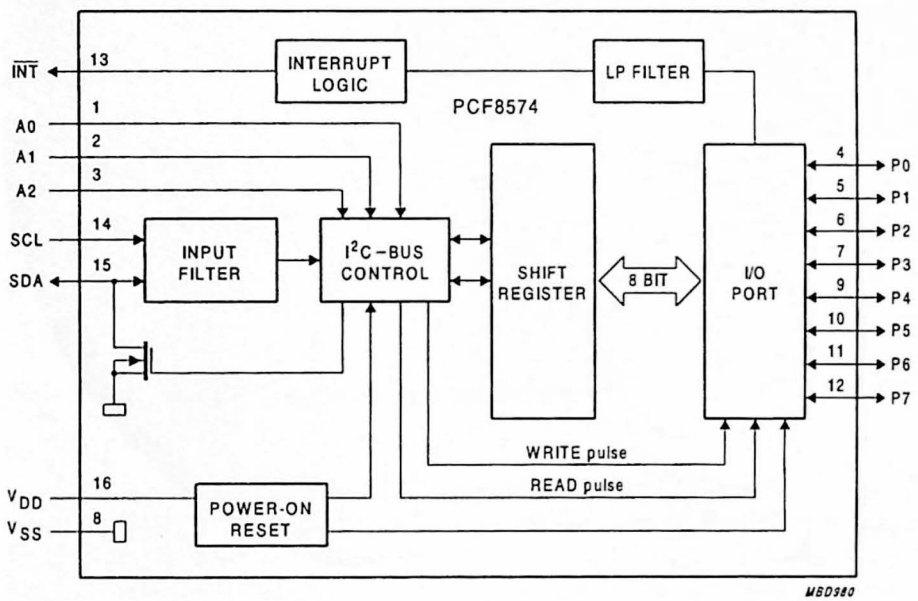


Abb. 12.5: Prinzipschaltbild eines Ausgangsports (Philips)

Der PCF8574 stellt einen Interrupt-Ausgang mit offenem Drain zur Verfügung, der an einen entsprechenden Interrupt-Eingang eines Mikrocontrollers angeschlossen werden kann. Ein Interrupt wird durch einen Pegelwechsel an einer der acht Portleitungen ausgelöst. Der Interrupt wird gelöscht, wenn der Port gelesen wird oder wenn der entsprechende Port in seinen Ausgangszustand zurückkehrt. Die Int-Ausgänge mehrerer Portexpander können zusammengeschaltet werden. Damit ergibt sich eine ODER-Verknüpfung, so dass der Mikrocontroller bei einem auftretenden Interrupt alle ICs abfragen muss, um die auslösende Portleitung zu ermitteln. Die hier vorgestellten Anwendungen arbeiten ohne den Interrupt. Die Eingangsleitungen werden stattdessen durch Polling beobachtet.

Das Testprogramm I2C_A.ASM demonstriert die Ausgabe von Daten an einen Portexpander PCF8574. Zur Kontrolle der Statusbits im Register I2CS im EZ-USB-Chip werden im Unterprogramm READY die Registerinhalte laufend an den Port C kopiert. Mit einem Oszilloskop kann hier das DONE-Bit beobachtet werden. Auch die Kontrolle des ACK-Bits ist hilfreich, wenn eine Verbindung über den I²C-Bus überprüft werden soll.

```
;I2C_A.asm
;Output to PCF8574

#include 8051.h
OEA  .equ   7F9Ch
OEB  .equ   7F9Dh
OEC  .equ   7F9Eh
PINSA .equ  7F99h
PINSB .equ  7F9Ah
PINSC .equ  7F9Bh
OUTA .equ   7F96h
OUTB .equ   7F97h
OUTC .equ   7F98h
I2CS  .equ  7FA5h
I2DAT .equ  7FA6h
START .equ  80h
STOP  .equ  40h
LASTRD .equ 20h
ACK   .equ  02h
DONE  .equ  01h

initPortC
     mov DPTR,#OEC
     mov A,#255
     movx @DPTR,A

I2COut
     mov   DPTR,#I2CS
     mov   a,#START
     movx  @DPTR,a
     mov   DPTR,#I2DAT
     mov   a,#40h      ;PCF8574 write
     movx  @DPTR,a
     lcall Ready
     mov   DPTR,#I2DAT
     mov   a,#85       ;data
     movx  @DPTR,a
     lcall Ready
     mov   DPTR,#I2DAT
     mov   a,#85       ;data
     movx  @DPTR,a
     lcall Ready
```

```
     mov   DPTR,#I2CS
     mov   a,#STOP
     movx  @DPTR,a
     lcall Ready
     lcall Delay
     sjmp  I2COut

Ready
     mov   DPTR,#I2CS
     movx  a,@DPTR
     mov DPTR,#OUTC
     movx @DPTR,A
     anl   a,#DONE
     cjne  a,#DONE,Ready
     ret

Delay
     mov   r7,#4
S1   mov   r6,#100
S2   djnz  r6,S2
     djnz  r7,S1
     ret

.End
```

Listing 12.1: Ausgabe von Daten über den I²C-Bus

Das Programm I2C_B.ASM zeigt Lesezugriffe auf den PCF8574. Die gelesenen Portdaten werden hier am Port C des EZ-USB wieder ausgegeben. Auf die Ausgabe des I2CS-Registers wird hier verzichtet.

```
;I2C_B.asm
;Input from PCF8574
:Output to Port C

#include 8051.h
OEA  .equ   7F9Ch
OEB  .equ   7F9Dh
OEC  .equ   7F9Eh
PINSA .equ  7F99h
PINSB .equ  7F9Ah
PINSC .equ  7F9Bh
OUTA .equ   7F96h
OUTB .equ   7F97h
OUTC .equ   7F98h
I2CS  .equ  7FA5h
I2DAT .equ  7FA6h
START .equ  80h
```

```
STOP  .equ  40h
LASTRD .equ 20h
ACK   .equ  02h
DONE  .equ  01h

initPortC
     mov DPTR,#0EC
     mov A,#255
     movx @DPTR,A

I2CIn
     mov   DPTR,#I2CS
     mov   a,#START
     movx  @DPTR,a
     mov   DPTR,#I2DAT
     mov   a,#41h      ;PCF8574 read
     movx  @DPTR,a
     lcall Ready
     mov   DPTR,#I2DAT
     movx  a,@DPTR     ;dummy byte
     lcall Ready
     mov   DPTR,#I2DAT
     movx  a,@DPTR     ;data byte
     mov DPTR,#OUTC
     movx @DPTR,A
     lcall Ready
     mov   DPTR,#I2CS
     mov   a,#LASTRD
     movx  @DPTR,a
     mov   DPTR,#I2DAT
     movx  a,@DPTR     ;dummy byte
     lcall Ready
     mov   DPTR,#I2CS
     mov   a,#STOP
     movx  @DPTR,a
     lcall Ready
     lcall Delay
     sjmp  I2CIn

 Ready
     mov   DPTR,#I2CS
     movx  a,@DPTR
     anl   a,#DONE
     cjne  a,#DONE,Ready
     ret

 Delay
     mov   r7,#4
 S1  mov   r6,#100
```

```
S2   djnz  r6,S2
     djnz  r7,S1
     ret

.End
```

Listing 12.2: Lesezugriffe auf den I²C-Bus

Der I²C-Bus bietet den großen Vorteil der leichten Erweiterbarkeit. Mit mehreren Portexpandern lassen sich zahlreiche zusätzliche Ports gewinnen. Bei drei verdrahtbaren Adressen des PCF8574 können bis zu acht Byteports gebildet werden. Mit dem PCF8574A steht ein kompatibler Baustein mit geänderter Basisadresse (70h) zur Verfügung, so dass noch einmal acht Byteports hinzukommen können.

Das Programm I2C_C.Asm zeigt die Verwendung eines Portbausteins für einen direkten Zugriff über den USB. Zur Übergabe von Daten dienen auch hier wieder bestimmte RAM-Adressen. Adresse 200h dient zur Ausgabe, Adresse 201h zur Eingabe.

```
;I2C_C.asm
;USB I2C bridge
:Output/Input 200h/201h

#include 8051.h
OEA  .equ   7F9Ch
OEB  .equ   7F9Dh
OEC  .equ   7F9Eh
PINSA .equ  7F99h
PINSB .equ  7F9Ah
PINSC .equ  7F9Bh
OUTA .equ   7F96h
OUTB .equ   7F97h
OUTC .equ   7F98h
I2CS  .equ  7FA5h
I2DAT .equ  7FA6h
START .equ  80h
STOP  .equ  40h
LASTRD .equ 20h
ACK   .equ  02h
DONE  .equ  01h

initPortC
     mov DPTR,#OEC
     mov A,#255
     movx @DPTR,A
I2COut
     mov   DPTR,#I2CS
```

```
      mov   a,#START
      movx  @DPTR,a
      mov   DPTR,#I2DAT
      mov   a,#40h      ;PCF8574 write
      movx  @DPTR,a
      lcall Ready
      mov   DPTR,#0200h
      movx  a,@DPTR     ;RAM data
      mov   DPTR,#I2DAT
      movx  @DPTR,a     ;write data
      lcall Ready
      mov   DPTR,#I2CS
      mov   a,#STOP
      movx  @DPTR,a
      lcall Ready
      lcall Delay

I2CIn
      mov   DPTR,#I2CS
      mov   a,#START
      movx  @DPTR,a
      mov   DPTR,#I2DAT
      mov   a,#41h      ;PCF8574 read
      movx  @DPTR,a
      lcall Ready
      mov   DPTR,#I2DAT
      movx  a,@DPTR     ;dummy byte
      lcall Ready
       mov   DPTR,#I2CS
      mov   a,#LASTRD
      movx  @DPTR,a
      mov   DPTR,#I2DAT
      movx  a,@DPTR     ;data byte
      mov   DPTR,#0201h
      movx  @DPTR,a     ;data to 201h
      lcall Ready
      mov   DPTR,#I2CS
      mov   a,#STOP
      movx  @DPTR,a
      lcall Ready
      lcall Delay
      sjmp  I2COut

Ready
      mov   DPTR,#I2CS
      movx  a,@DPTR
      anl   a,#DONE
      cjne  a,#DONE,Ready
      ret
```

```
Delay
     mov   r7,#4
S1   mov   r6,#100
S2   djnz  r6,S2
     djnz  r7,S1
     ret

.End
```

Listing 12.3: Eine USB-Brücke zum I²C-Bus

Das Assemblerprogramm kann mit einem kleinen Delphi-Programm nach Listing 12.4 getestet werden. Hier wird die Firmware geladen und gestartet. In einer Timer-Routine werden laufend Zustände des Schiebereglers über die Adresse 200h ausgegeben und Eingangs-Portzustände über 201h zurückgelesen.

```
unit PCF8574;

interface

uses  EZUSB,
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    ScrollBar1: TScrollBar;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Timer1: TTimer;
    procedure FormCreate(Sender: TObject);
    procedure ScrollBar1Change(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  private
    { Private-Deklarationen}
  public
    { Public-Deklarationen}
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ProgReset;
  DownLoadBin ('I2C_c.obj');
  WrRAM ($0200,0);
  ProgStart;
end;

procedure TForm1.ScrollBar1Change(Sender: TObject);
begin
  Label1.Caption := FloatToStr (Scrollbar1.Position)
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  WrRAM ($0200,Scrollbar1.Position);
  Label4.Caption := FloatToStr(RdRAM ($0201));
end;

end.
```

Listing 12.4: Delphi-Programm zur Datenübertragung

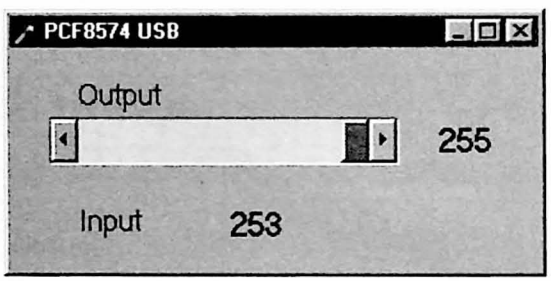


Abb. 12.6: Input und Output über den I²C-Bus

Abb. 12.6 zeigt das laufende Delphi-Programm zum Datenaustausch mit dem Portexpander. Die ausgegebenen Daten 255 setzen alle Portleitungen hoch. Damit stehen sie als Eingänge bereit. Gelesen wird der reale Zustand des Ports mit 253. Daraus ist erkennbar, dass die Leitung D1 mit der Wertigkeit 2 an Masse gelegt war.

12.4 I²C-EEPROMs

Serielle I²C-EEPROMs im 8-poligen Gehäuse werden in Größen von 16 Bytes bis 64 KB hergestellt. Der hardwaremäßig realisierte I²C-Bus des EZ-USB-Chips kann verwendet werden, um solche EEPROMs zu programmieren und auszulesen. Die

Speicher werden je nach Größe des internen Adressregisters unterschiedlich angesprochen. Kleine EEPROMs bis 256 Bytes verwenden einen 1-Byte-Adresszähler, größere zwischen 4 K und 64 K benötigen einen 2-Byte-Adresszähler. Alle EEPROMs verwenden die Adresse A0h als I²C-Basisadresse. Diese Adresse gilt, wenn alle herausgeführten Leitungen A0 bis A2 an Masse gelegt werden.

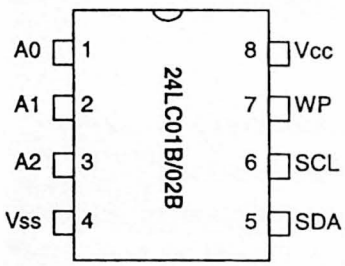


Abb. 12.7: Pinbelegung von I²C-EEPROMs (Quelle: Microchip)

Abb. 12.7 zeigt die Pinbelegung. Man erkennt die I²C-Leitungen SCL und SDA und die Adressleitungen A0 bis A2. Je nach Typ ist eine Leitung WP (Write Protect) vorhanden, mit der man Schreibzugriffe hardwaremäßig unterbinden kann. Für den direkten Anschluss an das EZ-USB-System mit einer Betriebsspannung von 3,3 V sollte man die Low-Voltage-Typen 24LCxx verwenden. Möglich ist aber auch der Einsatz von 24Cxx-Typen bei einer Betriebsspannung von 5 V. Die Leitungen SCL und SDA kommen auch dann mit 3,3-V-Pegeln aus.

24LC00: 16 Bytes
24LC01: 128 Bytes
24LC02: 256 Bytes

Die kleineren EEPROMs bis zu 256 Bytes (2 Kilobit) verwenden einen internen Adresszeiger mit einer Länge von 8 Bits, der nach der I²C-Busadresse übertragen werden muss. Das einfache Protokoll lässt daher nur Größen bis 256 Bytes zu. Es können bis zu 8 ICs gemeinsam verwendet werden, da 8 Busadressen zwischen A0h und AEh verwendet werden können. Die maximale Gesamtgröße beträgt daher 2 Kilobyte. Die größeren EEPROMs 24LC04 bis 24LC16 belegen jeweils mehrere I²C-Busadressen. Ein 24LC16 verhält sich daher exakt wie 8 im Adressraum untergebrachte 24LC02. Daher sind die Adress-Eingänge A0 bis A2 nicht belegt.

24LC04: 512 Bytes
24LC08: 1 Kilobyte
24LC16: 2 Kilobyte

Diese EEPROM-Typen müssen in Blöcken von 256 Bytes gelesen und beschrieben werden, wobei jedesmal eine neue Adressierung erfolgen muss. Sie können daher

nicht für den weiter unten beschriebenen EEPROM-Bootvorgang eingesetzt werden.

Die größeren EEPROMs über 2 Kilobyte verwenden ein geändertes Übertragungsprotokoll mit einem 16-Bit-Adresszeiger. Nach der Busadresse müssen insgesamt zwei weitere Bytes zur Angabe der internen Byte-Adresse gesendet werden. Das Protokoll ist also nicht mehr kompatibel zu den kleineren EEPROMs.

24LC32: 4 Kilobytes
24LC64: 8 Kilobytes
24LC128: 16 Kilobytes
24LC256: 32 Kilobytes

Hier soll ein möglichst universelles Programmiergerät für verschiedene EEPROMs realisiert werden. *Abb. 12.8* zeigt das fertige Delphi-Programm. Der Anwender kann den Typ des EEPROMs wählen und eine I²C-Adresse angeben. Es können mehrere ICs im Adressraum ab 160 (=A0h) angeschlossen sein. Zusätzlich ist eine Startadresse anzugeben, ab der Daten gelesen oder programmiert werden sollen.

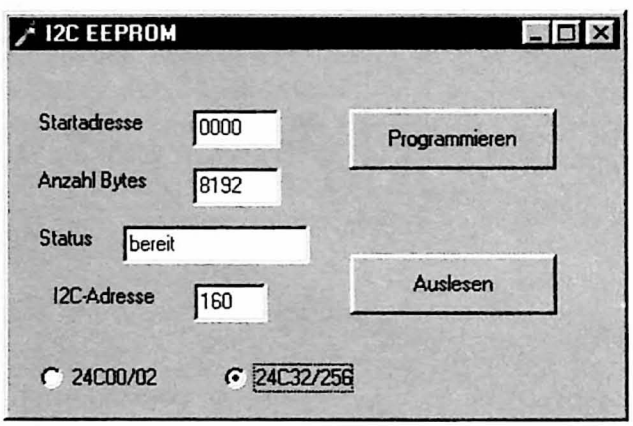


Abb. 12.8: Das universelle Programm EEPROM.EXE

Das Programm basiert auf einem direkten Austausch mit Binärdateien. Zum Programmieren gibt man eine Datei als Quelle der Daten an. Die Daten werden dann vollständig in das EEPROM übertragen. Zum Auslesen muss ebenfalls eine Datei angegeben werden. Es wird dann die im Fenster „Anzahl Bytes" eingetragene Datenmenge ausgelesen und gespeichert.

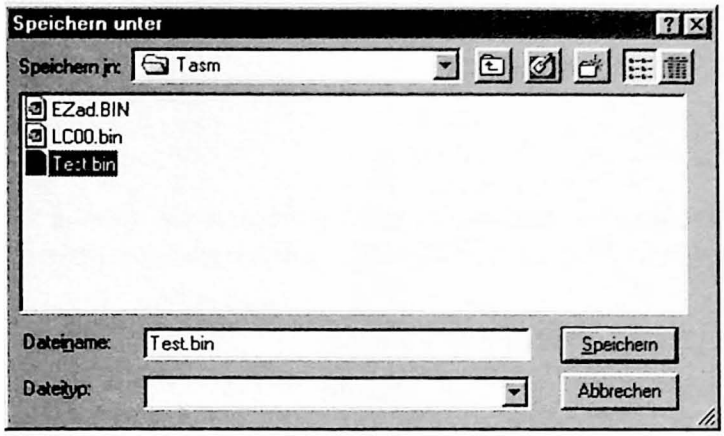


Abb. 12.9: Dateifenster zum Speichern von EEPROM-Daten

Zum Programmieren eines Bytes sind folgende Schritte nötig:

- I²C-Start
- Senden der I²C-Adresse (A0h)
- Nur ab 4K: Senden des Highbytes der internen Adresse
- Senden des Lowbytes der internen Adresse
- Senden des zu programmierenden Bytes
- I²C-Stop
- Abwarten der Programmierzeit (ca. 1 ms)

Je nach Typ kann mehr als ein Byte in einem Block programmiert werden. Von diesem zeitsparenden Block-Write-Modus wird hier kein Gebrauch gemacht, weil das Programm möglichst universell sein soll und auch zu älteren EEPROM-Typen passen soll.

Zum Auslesen eines Bytes sind folgende Schritte erforderlich:

- I²C-Start
- Senden der I²C-Adresse (A0h)
- Nur ab 4K: Senden des Highbytes der internen Adresse
- Senden des Lowbytes der internen Adresse
- I²C-Stop

- I²C-Start
- Senden der I²C-Leseadresse (A1h)
- Lesen beliebig vieler Bytes
- I²C-Stop

Das Programm EEPROM.PAS verwendet vier Assembler-Programme, die zum Lesen und Programmieren in das RAM des EZ-USB-Chips geladen und dort gestartet werden:

- I2EErd_1.am: Auslesen von 256 Bytes, 1-Byte-Adresse
- I2EErd_2.am: Auslesen von 256 Bytes, 2-Byte-Adresse
- I2EEwr_1.asm: Programmieren eines Bytes, 1-Byte-Adresse
- I2EEwr_2.asm: Programmieren eines Bytes, 2-Byte-Adresse

Der Datenaustausch zwischen Prozessor und PC-Programm findet über das RAM statt. Im Bereich 1FDh bis 1FFh werden Adress-Daten an das Programm übergeben, im Bereich 200h bis 2FFh werden Nutzdaten übergeben.

1FDh: I²C-Basisadresse (Default 160=A0h)
1FEh: Startadresse Highbyte
1FFh: Startadresse Lowbyte
200h: Datenbyte beim Programmieren
200h...2FFh: 256 gelesene Bytes beim Auslesen

Das Programm I2CEEwr_1.asm zeigt das Programmieren eines Bytes. Eine spezielle Pause für die Programmierzeit muss hier nicht eingehalten werden, weil dies im aufrufenden Delphi-Programm erfolgt. Das Programm überträgt nur die Lowbyte-Adresse aus der RAM-Adresse 1FFh, da es für kleine EEPROMs ausgelegt ist.

```
;I2EEwr_1.asm
;Write EEPROM 24C00
;I2C-Adr 1FDh
;Adr/Dat 1FEh,1FFh/200h

#include 8051.h
OEA  .equ   7F9Ch
OEB  .equ   7F9Dh
OEC  .equ   7F9Eh
PINSA .equ  7F99h
PINSB .equ  7F9Ah
PINSC .equ  7F9Bh
OUTA .equ   7F96h
OUTB .equ   7F97h
OUTC .equ   7F98h
```

```
I2CS  .equ  7FA5h
I2DAT .equ  7FA6h
START .equ  80h
STOP  .equ  40h
LASTRD .equ 20h
ACK   .equ  02h
DONE  .equ  01h
DPS   .equ  86h

I2COut
     mov   DPTR,#I2CS
     mov   a,#START
     movx  @DPTR,a
     mov   DPTR,#01FDh ;I2C write address
     movx  a,@DPTR
     mov   DPTR,#I2DAT
     mov   a,#0A0h     ;24C256 write
     movx  @DPTR,a
     lcall Ready
     mov   DPTR,#01FFh
     movx  a,@DPTR     ;start address Low
     mov   DPTR,#I2DAT
     movx  @DPTR,a     ;write
     lcall Ready
     mov   DPTR,#0200h
     movx  a,@DPTR     ;data
     mov   DPTR,#I2DAT
     movx  @DPTR,a     ;write data
     lcall Ready
     mov   DPTR,#I2CS
     mov   a,#STOP
     movx  @DPTR,a
     lcall Ready
End  sjmp  End

Ready
     mov   DPTR,#I2CS
     movx  a,@DPTR
     anl   a,#DONE
     cjne  a,#DONE,Ready
     ret
.End
```

Listing 12.5: Programmieren eines EEPROM-Bytes

Zur Programmierung großer EEPROMs wird in I2EEwr_2.asm auch das Highbyte der Adresse übertragen.

```
;I2EEwr_2.asm
...
     mov   DPTR,#01FEh
     movx  a,@DPTR      ;start address High
     mov   DPTR,#I2DAT
     movx  @DPTR,a      ;write
     lcall Ready
     mov   DPTR,#01FFh
     movx  a,@DPTR      ;start address Low
     mov   DPTR,#I2DAT
     movx  @DPTR,a      ;write
     lcall Ready
     mov   DPTR,#0200h
     movx  a,@DPTR      ;data
     mov   DPTR,#I2DAT
     movx  @DPTR,a      ;write data
     lcall Ready
  ...
```

Listing 12.6: Auszug aus dem Programm I2EEwr_2.asm

Das Programm zum Auslesen von Daten ist etwas umfangreicher, weil hier das IC einmal in Schreibrichtung und einmal in Leserichtung adressiert wird. Das eigentliche Lesen erfolgt in einer Zählschleife. Es werden grundsätzlich 256 Bytes ausgelesen und in den Adressbereich ab 200h kopiert. Kleine EEPROMs wie der 24LC00 mit 16 Bytes werden durch Überlauf des internen Adresszeigers tatsächlich mehrfach ausgelesen, wobei sich der Datenbereich mehrfach spiegelt. Für größere EEPROMs mit mehr als 256 Bytes muss das Programm mehrfach neu gestartet werden, um den gesamten Datenbereich in Blöcken auszulesen. Listing 12.7 zeigt das Leseprogramm für große EEPROMs mit 2-Byte-Adressen.

```
;I2EErd_2.asm
;Read EEPROM 24C256
;I2C Adr 1FDh
:Adr/Dat 1FEh,1FFh/200h...2FFh

#include 8051.h
OEA  .equ   7F9Ch
OEB  .equ   7F9Dh
OEC  .equ   7F9Eh
PINSA .equ  7F99h
PINSB .equ  7F9Ah
PINSC .equ  7F9Bh
OUTA .equ   7F96h
OUTB .equ   7F97h
OUTC .equ   7F98h
I2CS  .equ  7FA5h
```

```
I2DAT .equ  7FA6h
START .equ  80h
STOP  .equ  40h
LASTRD .equ 20h
ACK   .equ  02h
DONE  .equ  01h
DPS   .equ  86h

I2COut
     mov   DPTR,#I2CS
     mov   a,#START
     movx  @DPTR,a
     mov   DPTR,#01FDh ;I2C write address
     movx  a,@DPTR
     mov   DPTR,#I2DAT
     movx  @DPTR,a
     lcall Ready
     mov   DPTR,#01FEh
     movx  a,@DPTR     ;start address high
     mov   DPTR,#I2DAT
     movx  @DPTR,a     ;write data
     lcall Ready
     mov   DPTR,#01FFh
     movx  a,@DPTR     ;start address low
     mov   DPTR,#I2DAT
     movx  @DPTR,a     ;write data
     lcall Ready
     mov   DPTR,#I2CS
     mov   a,#STOP
     movx  @DPTR,a
     lcall Ready
     lcall Delay

I2CIn
     mov DPS,#1   ;select DPTR1
     mov DPTR,#200h      ;RAM data
     mov DPS,#0   ;select DPTR0
     mov   DPTR,#I2CS
     mov   a,#START
     movx  @DPTR,a
     mov   DPTR,#01FDh  ;I2C address
     movx  a,@DPTR
     inc   a            ;read address
     mov   DPTR,#I2DAT
     movx  @DPTR,a
     lcall Ready
     mov   DPTR,#I2DAT
     movx  a,@DPTR      ;dummy byte
     lcall Ready
     mov   r7,#00h
```

```
Loop
     mov   DPTR,#I2DAT
     movx  a,@DPTR       ;data byte
     inc   DPS
     movx  @DPTR,a       ;save data
     inc   DPTR
     inc   DPS
     lcall Ready
     djnz  r7,Loop       ;256 bytes
     mov   DPTR,#I2CS
     mov   a,#LASTRD
     movx  @DPTR,a
     mov   DPTR,#I2DAT
     movx  a,@DPTR       ;dummy byte
     lcall Ready
     mov   DPTR,#I2CS
     mov   a,#STOP
     movx  @DPTR,a
     lcall Ready
     lcall Delay
End  sjmp  End

Ready
     mov   DPTR,#I2CS
     movx  a,@DPTR
     anl   a,#DONE
     cjne  a,#DONE,Ready
     ret

Delay
     mov   r7,#4
S1   mov   r6,#100
S2   djnz  r6,S2
     djnz  r7,S1
     ret
.End
```

Listing 12.7: Auslesen von 256 Bytes mit Übertragung einer 2-Byte-Adresse

Das Delphi-Programm EEPROM.Pas lädt beim Lesen und beim Programmieren jeweils das passende Assemblerprogramm für kleine oder für große EEPROMs in das RAM des EZ-USB-Chips. Beim Programmieren und beim Auslesen selbst wird das geladene Programm dann jeweils mit den erforderlichen Adress-Daten versorgt und einmal gestartet und nach kurzer Wartezeit wieder gestoppt.

Das Programmieren mit Einzelbytes ist relativ langsam, da jedesmal die volle Adresse mit Highbyte und Lowbyte und nur ein Datenbyte übertragen werden. Damit ist

für unterschiedliche EEPROM-Typen eine ausreichende Programmierzeit garantiert. Wenn man einen bestimmten EEPROM-Typ verwenden möchte, kann das Programmieren durch Verwendung des Block-Write-Modus und durch Kontrolle der Programmierzeit im Assemblerprogramm verbessert werden.

Das Auslesen von EEPROMs ist relativ schnell, weil jeweils 256 Bytes in einem Block übertragen werden. Bei größeren Datenmengen werden mehrere Blöcke angefordert, wobei jedesmal die neuen Anfangsadressen in den internen Adresszeiger des EEPROMs geladen werden.

```
unit EEprom;

interface

uses  EZUSB,
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Label1: TLabel;
    Button1: TButton;
    Button2: TButton;
    RadioButton1: TRadioButton;
    RadioButton2: TRadioButton;
    OpenDialog1: TOpenDialog;
    SaveDialog1: TSaveDialog;
    Edit2: TEdit;
    Label2: TLabel;
    Edit3: TEdit;
    Label3: TLabel;
    Edit4: TEdit;
    Label4: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure RadioButton2Click(Sender: TObject);
    procedure RadioButton1Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Edit4Change(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

```
  Typ256: Boolean;
  Adresse, Anzahl: Integer;

implementation

{$R *.DFM}

procedure Brennen (Adresse: Word; Wert: Byte);
var n, m: Integer;
begin
      WrRAM ($01FE,Hi(Adresse));
      WrRAM ($01FF,Lo(Adresse));
      WrRAM ($0200,Wert);
      ProgStart;
      Sleep (2);
      ProgReset;
end;


procedure TForm1.FormCreate(Sender: TObject);
begin
  Typ256 := True;
end;

procedure TForm1.Button1Click(Sender: TObject);
 VAR  f :file of byte;
      r,wert :Byte;
      Adresse, I2CAdr, n, Code : Integer;
begin
   val (Edit1.Text, Adresse, Code);
   val (Edit4.Text, I2CAdr, Code);
   I2CAdr := I2CAdr and 254;
   WrRAM ($01FD,Lo(I2CAdr));
   OpenDialog1.FileName := '*.bin';
   OpenDiaLog1.Execute;
   if OpenDialog1.FileName > '' then begin
     AssignFile(f,OpenDialog1.Filename);
    {$I-} Reset(f); {$I+}
    r:=IOResult;
    IF r = 0 then begin
      if Typ256 then DownLoadBin ('I2EEwr_2.obj')
      else DownLoadBin ('I2EEwr_1.obj');
      for n:= 1 to filesize (f) do begin
        Read(f,wert);
        Brennen(Adresse+n-1,Wert);
      end;
      CloseFile(f);
      Edit3.Text := 'Brennen OK'
    end;
  end;
```

```
end;

procedure TForm1.Button2Click(Sender: TObject);
 VAR  f :file of byte;
      r,wert,Block :Byte;
      Adresse, I2CAdr, Anzahl,n, Code : Integer;
begin
   val (Edit1.Text, Adresse, Code);
   val (Edit2.Text, Anzahl, Code);
   val (Edit4.Text, I2CAdr, Code);
   I2CAdr := I2CAdr and 254;
   WrRAM ($01FD,Lo(I2CAdr));
   SaveDialog1.FileName := '*.bin';
   SaveDiaLog1.Execute;
   if SaveDialog1.FileName > '' then begin
     AssignFile(f,SaveDialog1.Filename);
    {$I-} Rewrite(f); {$I+}
    r:=IOResult;
    IF r = 0 then begin
      Block := 0;
      ProgReset;
      if Typ256 then DownLoadBin ('I2EErd_2.obj')
          else DownLoadBin ('I2EErd_1.obj');
      repeat
      WrRAM ($01FE,Hi(Adresse + Block * 256));
      WrRAM ($01FF,Lo(Adresse + Block * 256));
      Block := Block + 1;
      ProgStart;
      Sleep (100);
      ProgReset;
      ReadRAMbytes ($200,256);
      if Anzahl > 256 then begin
        for n:= 0 to 255 do begin
          Wert := InBuffer[n];
          write (f,Wert);
        end;
        Anzahl := Anzahl -256;
      end else begin
        for n:= 0 to (Anzahl-1) do begin
          Wert := InBuffer[n];
          write (f,Wert);
        end;
        Anzahl := 0;
      end;
      until Anzahl = 0;
      CloseFile(f);
      end;
      Edit3.Text := 'OK';
  end;
```

```
end;

procedure TForm1.RadioButton2Click(Sender: TObject);
begin
  Typ256 := false;
end;

procedure TForm1.RadioButton1Click(Sender: TObject);
begin
  Typ256 := true;
end;
end.
```

Listing 12.8: Das Delphi-Programm EEPROM.PAS

12.5 Der EEPROM-Bootloader

Der EZ-USB-Chip erleichtert die Entwicklung eigener USB-Anwendungen, weil der USB-Kern bereits ohne eine zusätzliche Firmware ein Default-Interface zur Verfügung stellt und sich selbständig enumeriert. Alle Device-Deskriptoren sind intern vorhanden und werden automatisch gemeldet. Der Chip stellt daher bereits ein vollständiges Entwicklungssystem dar, in das man 8051-Software laden kann.

Ein über den USB geladenes Programm kann sich erneut am USB anmelden und dabei eigene Deskriptoren verwenden. Auf diese Weise kann ein Anwender z.B. die firmeneigene Vendor-ID melden, um einen eigenen Treiber zu laden. Der EZ-USB-Chip stellt dazu ein besonderes Verfahren bereit, das als ReNumeration bezeichnet wird. Der Chip simuliert dazu eine Trennung vom Bus durch Abschalten des Pullup-Widerstands an der Datenleitung D+. Das Betriebssystem entfernt daraufhin den vorhandenen Treiber aus dem Speicher. Danach meldet sich der Chip wieder an, wobei diesmal das interne ReNum-Bit gesetzt ist. Nun meldet sich der Chip mit den von der Firmware bereitgestellten Deskriptoren erneut an.

Eine eigene Vendor-ID kann auch beim Start des Systems automatisch aus einem angeschlossenen I²C-EEPROM geladen werden. Damit hat man die Möglichkeit, gleich beim ersten Start eigene Einstellungen und Deskriptoren zu verwenden. Dies ist erforderlich, wenn ein firmeneigener Treiber geladen werden soll.

Der EZ-USB-Chip erkennt automatisch EEPROMs bei Adresse A0h und A2h. Da es keine Möglichkeit gibt, automatisch den EEPROM-Typ zu erkennen, wurde festgelegt, dass kleine EEPROMs mit 1-Byte-Adresse bei I²C-Adresse A0h liegen müssen, große EEPROMs mit 2-Byte-Adresse dagegen bei A2h, d.h. die externe Adressleitung A0 des EEPROMS muss hoch gelegt werden.

Wenn nach einem Reset ein externes EEPROM erkannt wurde, wird zunächst das erste Byte gelesen. Nur wenn es den Wert B0h oder B2h aufweist, werden die weiteren Daten beachtet. Alle anderen Inhalte werden ignoriert, so dass das EEPROM für andere Zwecke zur Verfügung steht.

Wenn das EEPROM an der ersten Stelle das Byte B0h hat, liest das System die folgenden sechs Bytes und ersetzt damit die Vendor ID, die Produkt ID und die Device ID im USB-Kern. Man kann also mit geringstem Aufwand die eigenen ID-Nummern verwenden. Der Aufbau der Daten ist in Tabelle 12.2 dargestellt.

Tabelle 12.2: Aufbau eines Start-EEPROMs mit Startbyte B0h

EPROM Adresse	Daten
0	B0h
1	Vendor ID Lowbyte (47h)
2	Vendor ID Highbyte (05, Anchor Chips)
3	Product ID Lowbyte (31)
4	Product ID Highbyte (21, EZ-USB)
5	Device ID Lowbyte (??))
6	Device ID Highbyte

Der folgende Versuch zeigt, dass es möglich ist, durch entsprechende Inhalte des EEPROMs ein anderes Gerät anzumelden. Dies hat nur Sinn, wenn auch ein passender Treiber vorliegt. *Abb. 12.10* zeigt die Binärdatei mit den IDs des Cypress-Thermometers. Diese Daten wurden zur Probe in eine EEPROM 24LC00 an der Adressse A0h programmiert. Dann wurde das Gerät vom Bus getrennt und erneut verbunden.

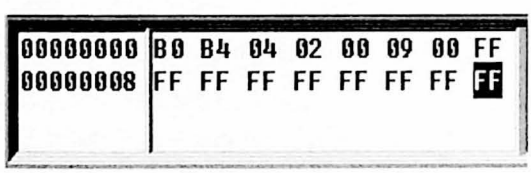


Abb. 12.10: Die IDs zum Anmelden eines Cypress-Thermometers

Der Versuch zeigt, dass tatsächlich der Thermometer-Treiber geladen werden kann. Offensichtlich spielt es keine Rolle, dass das Thermometer ein Lowspeed-Gerät ist, während das EZ-USB-System mit dem Fullspeed-USB arbeitet. Natürlich kann man nun nicht wirklich auf das Gerät zugreifen, da die entsprechende Firmware fehlt.

Man kann aber in der Systemsteuerung nachsehen, welches Gerät angemeldet wurde. *Abb. 12.11* zeigt den Eintrag des Thermometers. Ein anderes USB-Gerät ist nicht vorhanden.

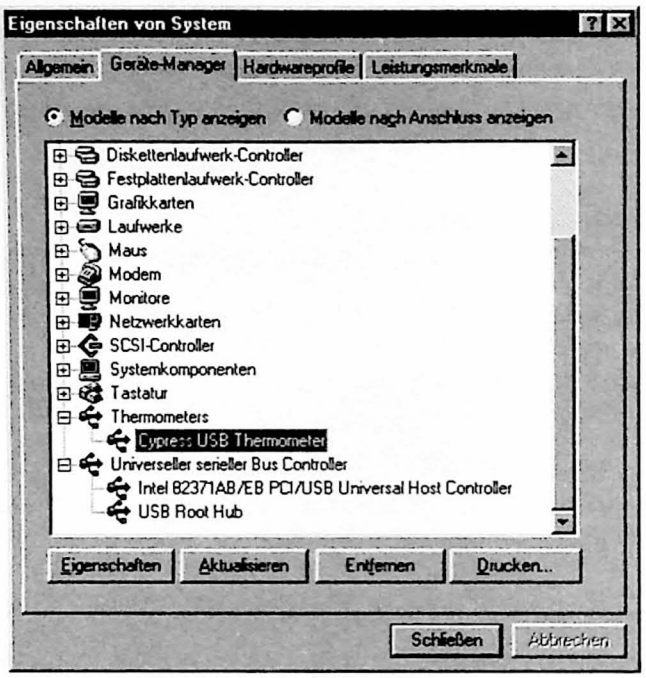


Abb. 12.11: Der geladene Thermometer-Treiber

Nach diesem Versuch kann das EEPROM nicht einfach im System umprogrammiert werden, da der EZ-USB-Treiber fehlt. Um das System wieder neu zu starten, muss zunächst das EEPROM aus dem Sockel gezogen werden. Nach einer neuen Anmeldung des EZ-USB-Systems kann man neue Daten programmieren. Es reicht, das erste Byte B0h zu ändern, damit die veränderten IDs beim Neustart nicht mehr geladen werden.

Der Anchor-USB-Kern stellt noch eine erweiterte Möglichkeit zur Verfügung, das System aus einem EEPROM zu initialisieren. Wenn das erste Byte B2h ist, wird außer den drei IDs auch ein komplettes 8051-Programm aus dem EEPROM ins intere RAM übertragen und gestartet. Gleichzeitig wird das Interne ReNum-Bit gesetzt, so dass der 8051 alle USB-Requests selbst bedienen muss, die bisher durch den USB-Kern behandelt wurden.

Tabelle 12.3: Aufbau eines Start-EEPROMs mit Startbyte B2h

EPROM Adresse	Daten
0	B2h
1	Vendor ID Lowbyte (47h)
2	Vendor ID Highbyte (05, Anchor Chips)
3	Product ID Lowbyte (31)
4	Product ID Highbyte (21, EZ-USB)
5	Device ID Lowbyte
6	Device ID Highbyte
7	Blocklänge Highbyte
8	Blocklänge Lowbyte
9	Startadresse Highbyte
10	Startadresse Lowbyte
...	Datenbereich des Blocks
...	weitere Datenblöcke
...	80h (letzter Block)
...	01h (Länge = 1)
...	7Fh (Adresse 7F92 = CPUCS)
...	92h
...	00h (CPUCS = 00h, Programm Start)

Das Programm kann in mehrere Datenblöcke aufgeteilt werden, wobei jeder Eintrag mit Länge und Startadresse angegeben wird. Der letzte Eintrag wird durch ein gesetztes höchstwertiges Bit im Highbyte der Längenangabe gekennzeichnet. Der letzte Datenblock ist normalerweise der Dateninhalt des CPUCS-Registers, also ein Block der Länge 1. Da es sich um den letzten Block handelt, wird statt 0001h die Länge 8001h angegeben. Das geladene Programm startet automatisch, sobald das Reset-Bit im CPUCS-Register gelöscht wird.

Die Anwendung ist nicht so einfach wie mit einem B0h-EEPROM, weil das geladene Programm alle USB-Requests selbst behandeln muss und diese Arbeit nicht mehr dem USB-Kern überlassen kann. Man muss bereits eine eigenständige USB-Anwendung haben, die dann automatisch aus dem EEPROM geladen werden kann. Wegen der relativ großen Programmlänge muss ein größeres EEPROM mit 2-Byte-Adresszeiger bei I²C-Adresse A2h eingesetzt werden.

13 Bulk-Transfer

Bisher wurde ausschließlich der Control-Transfer über den USB verwendet. Der Control-Transfer über Endpoint 0 ist die einfachste Art, Daten auszutauschen, weil jedes USB-Gerät ohne weitere Initialisierung auf Control-Transfers reagiert. Nach der USB-Philosophie sollte diese Betriebsart eigentlich nur zur Initialisierung eines Geräts eingesetzt werden. In Mess-, Steuerungs- und Regelungsanwendungen bietet sie jedoch auch im laufenden Betrieb erhebliche Vorteile. Control-Zugriffe arbeiten immer in beiden Datenrichtungen, so dass sich ein einfacher Austausch kleiner Datenmengen ergibt. Bulk-Transfers besitzen dagegen immer nur eine Datenrichtung. Sie bieten allerdings andere Vorteile.

Bulk-Transfers werden für die sichere Übertragung großer Datenmengen verwendet, die nicht zeitkritisch sind. Typische Beispiele sind Drucker und Scanner. Die gesamte Bandbreite des USB ist zu 90 % für isochronen und Interrupt-Verkehr reserviert und zu 10% für Control-Transfers. Für Bulk-Transfers ist keine Bandbreite reserviert. Falls also die Restbandbreite durch viele Geräte am Bus verbraucht wurde, werden Bulk-Transfers verzögert, bis wieder genügend Bandbreite vorhanden ist. Dies ist aber normalerweise kein Problem, da auch mit mehreren Geräten meist nur ein Bruchteil der Gesamtbandbreite belegt wird.

Mit dem Bulk-Transfer beginnt erst die „richtige" Arbeit mit dem USB. Es gibt mehr als einen möglichen Endpoint, das Gerät muss erst initialisiert werden, und der Bulk-Transfer ist wesentlich empfindlicher gegen Programmfehler, führt also schneller zu Abstürzen. Deshalb wird hier ein Neuansatz gemacht und die USB-Host-Software noch einmal von vorn aufgebaut. Vieles, was bisher nur „quick and dirty" programmiert wurde, soll nun noch einmal sauber implementiert werden. Um alle verfügbaren Treiberfunktionen aufrufen zu können, werden die entsprechenden C-Headerfiles in Delphi übertragen.

13.1 Pipes und Endpoints

Nach einem Neustart muss jedes USB-Gerät auf Control-Requests über den Endpoint 0 antworten. Hier werden alle Deskriptoren abgefragt, die bei der Enumeration benötigt werden. Außerdem können über Endpoint 0 bestimmte Initialisierungen durchgeführt werden, die zum Betrieb des Geräts nötig sind.

Ein neu enumeriertes Gerät hat normalerweise noch keine anderen Übertragungskanäle als Endpoint 0. Alle anderen Endpoints müssen erst eingeschaltet werden.

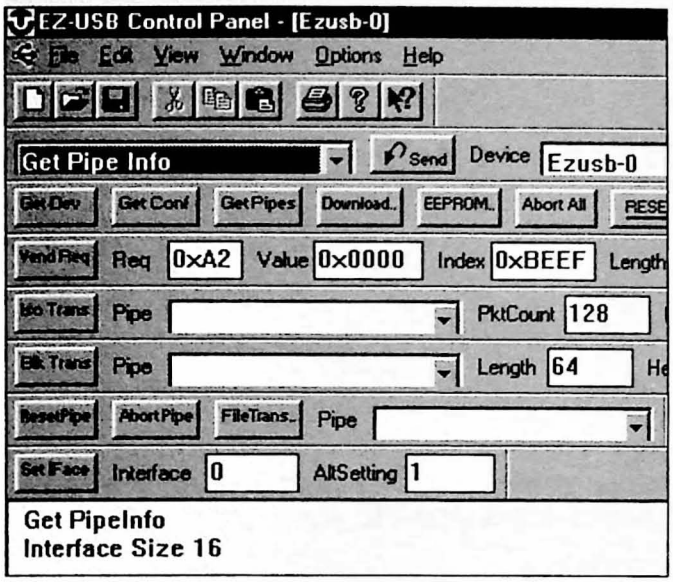


Abb. 13.1: Die Get-Pipe-Abfrage beim unkonfigurierten System

Ein USB-Gerät kann mehrere „Interfaces" enthalten, die jeweils mehrere „Configurations" unterstützen. Alle unterstützen Betriebsarten werden bei der Enumerierung abgefragt und sind dem System bekannt. Die Anwendersoftware oder ein Treiber kann nun eine der möglichen Konfigurationen unterstützen. Das Standard EZ-USB-Device unterstützt ein Interface 0 mit drei möglichen Konfigurationen. Über das Kommando „Set Interface" mit Interface =0 und AltSetting=1 konfiguriert man das System mit der Konfuguration 1. In dieser Einstellung verfügt es über einen Interrupt-Endpoint, sechs Bulk-Endpoints und sechs isochrone Endpoints. Jeder Endpoint ist einer Pipe zugeordnet.

Mit dem konfigurierten Interface können nun prinzipiell auch Bulk- und isochrone Transfers über das Control Panel durchgeführt werden. Für jede Aktion muss die gewünschte Pipe ausgesucht werden. Allerdings muss hierzu eine passende Firmware geladen und gestartet werden, denn die übertragenen Daten müssen auf der anderen Seite verarbeitet werden. Jeder Pipe entspricht ein Endpoint mit einem zugeordneten FIFO-Puffer. Die Firmware im 8051-System muss Puffer füllen und auslesen sowie entsprechende Steuerregister bedienen, um den USB-Kern über die Bereitschaft und die Menge der verarbeiteten Daten zu informieren. Nur so kann eine Kommunikation über eine Pipe tatsächlich ausgeführt werden.

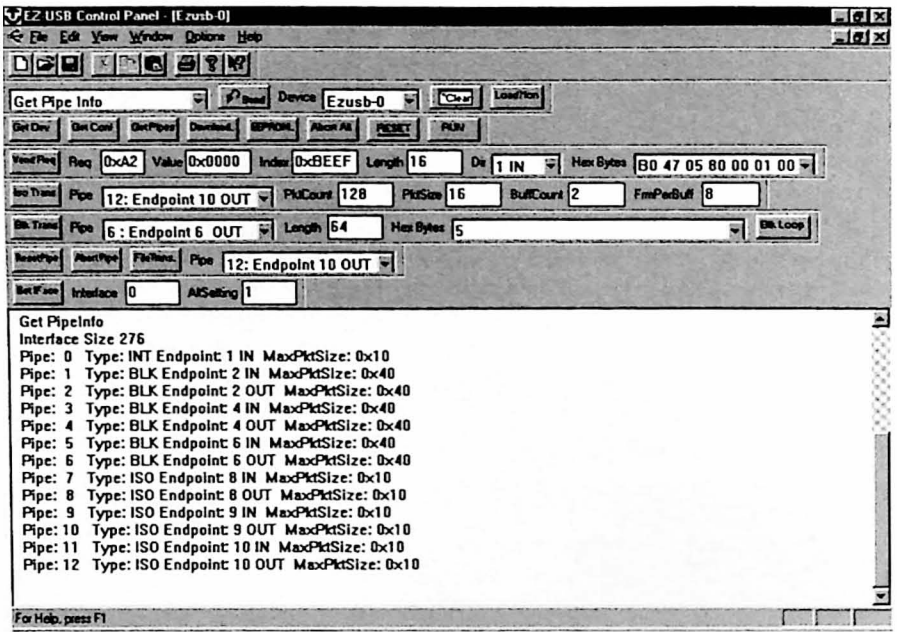


Abb. 13.2: Ergebnis von "Get Pipe Info" in der Konfiguration 1

Fehler bei der Verwendung der Kommandos Bulk Transfer und Iso Transfer führen leicht zu Systemabstürzen, die nur noch durch den Netzschalter zu beheben sind. Mögliche Fehler sind z.B. der Start eines Transfers ohne die passende Firmware oder der versuchte Zugriff auf eine nicht von der Firmware unterstützte Pipe.

13.2 Ein Assembler-Programm zum Bulk-Transfer

Das folgende Testprogramm zum Bulk-Transfer wurde aus den Beispielprogrammen des EZ-USB-Starterkits übernommen und für die Übersetzung mit TASM leicht angepasst. Die erforderlichen Register sind der Übersicht halber direkt in das Listing 13.1 eingetragen. Es wird nur Endpoint 2 verwendet. Je 64 Datenbytes werden gesendet und empfangen.

```
;BULK1.ASM
#include 8051.h

OEC   .equ   7F9Eh
PINSC .equ   7F9Bh
OUTC  .equ   7F98h
DPS   .equ   86h
```

```
EP2OUT    .equ 7DC0h
EP2IN     .equ 7E00h
EP2INCS   .equ 7FB8h
EP2INBC   .equ 7FB9h
EP2OUTCS  .equ 7FC8h
EP2OUTBC  .equ 7FC9h

start mov SP,#80 ; set stack
      mov dptr,#EP2IN ; fill EP2IN buffer with decrementing count
      mov r7,#64
fill  mov a,r7
      movx @dptr,a
      inc dptr
      djnz r7,fill
      ;
      mov r1,#0 ; INS counter
      mov r2,#0 ; OUTS counter
      mov dptr,#EP2INBC
      mov a,#40h
      movx @dptr,a ; arm the IN2 transfer
      ;
loop  mov dptr,#EP2INCS ; Control & Status reg
      movx a,@dptr
      jnb ACC.1,serviceIN2 ; not busy-service it
      mov dptr,#EP2OUTCS
      movx a,@dptr
      jb ACC.1,loop ; busy-keep looping
      ;
serviceOUT2
      inc r2 ; OUT packet counter
      mov dptr,#EP2OUTBC
      movx @dptr,a ; load bc=X to re-arm OUT2
      sjmp loop
      ;
serviceIN2
      inc r1 ; IN packet counter
      mov dptr,#EP2IN
      mov a,r1
      movx @dptr,a ; load IN count into first byte
      inc dptr ; second byte
      mov a,r2
      movx @dptr,a ; load OUT count into second byte
      mov dptr,#EP2INBC
      mov a,#40h
      movx @dptr,a ; load bc=64 to re-arm IN2
      sjmp loop
.end
```

Listing 13.1: Bulk-Datentransfer über EP2IN und EP2OUT

Das Programm füllt den EP2-IN-Buffer mit 64 absteigenden Bytes. Zwei Zähler in R1 und R2 werden auf Null gesetzt. Dann wird der EP2IN-Transfer für 64 Bytes vorbereitet, indem das EP2INBC-Register mit dem Wert 64 gefüllt wird.

Durch eine Polling-Abfrage wartet das Programm, bis bestimmte Flags im Control- und Statusregister EP2INCS oder EPOUTCS gesetzt sind. Es werden dann die entsprechenden Service-Routinen für IN- und OUT-Transfers angesprungen. Sobald über den EZ-USB-Kern ein IN-Transfer durchgeführt wird, muss der Puffer erneut vorbereitet werden und der entsprechende Byte-Zähler erneut gesetzt werden. Das Programm ersetzt die ersten beiden Bytes im Puffer jeweils durch die aktuellen Werte in R1 und R2, also durch die Anzahl der bereits durchgeführten IN- und OUT-Transfers.

Jeder OUT-Transfer wird in der Routine serviceOUT2 behandelt. Die empfangenen Daten werden aber nicht ausgewertet. Es wird nur der Zähler R2 erhöht und der nächste OUT-Transfer vorbereitet, indem EP2OUTBC erneut mit dem Wert 64 für 64 zu empfangende Bytes geladen wird.

Das Programm wird mit einem kleinen Delpi-Programm in das System geladen und gestartet:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ProgReset;
  DownLoadBin ('Bulk1.obj');
  ProgStart;
end;
```

Listing 13.2: Übertragung und Start der Firmware

Nun kann die Übertragung mit dem Control-Panel getestet werden. Für IN-Transfers kann nur Pipe 1 zum Puffer EP2IN verwendet werden, für OUT-Transfers nur Pipe 2 zum Puffer EP2OUT. Beim ersten Lesen erhält man die von der Firmware vorbereiteten 64 absteigenden Bytes. Beim Senden werden die voreingestellten Bytes 05h übertragen. Lese- und Sendezugriffe können in unregelmäßiger Weise aufeinander folgen. Vom zweiten Lesezugriff ab erkennt man in den ersten beiden Bytes die Zählerstände für erfolgte Lese- und Schreibzugriffe.

13.3 Host-Software zum Bulk-Transfer

Bisher wurden nur wenige Funktionen und Control-Codes des allgemeinen EZUSB-Treibers verwendet. Dabei war es sinnvoll, Control-Codes als feste Konstanten ein-

zusetzen. Dieses Verfahren wird aber fehlerträchtig, wenn sehr viele Funktionen eingesetzt werden sollen. Es ist daher besser, direkt von den vorliegenden Informationen des Starterkits auszugehen und die entsprechenden Konstanten-Deklarationen in Dephi umzusetzen.

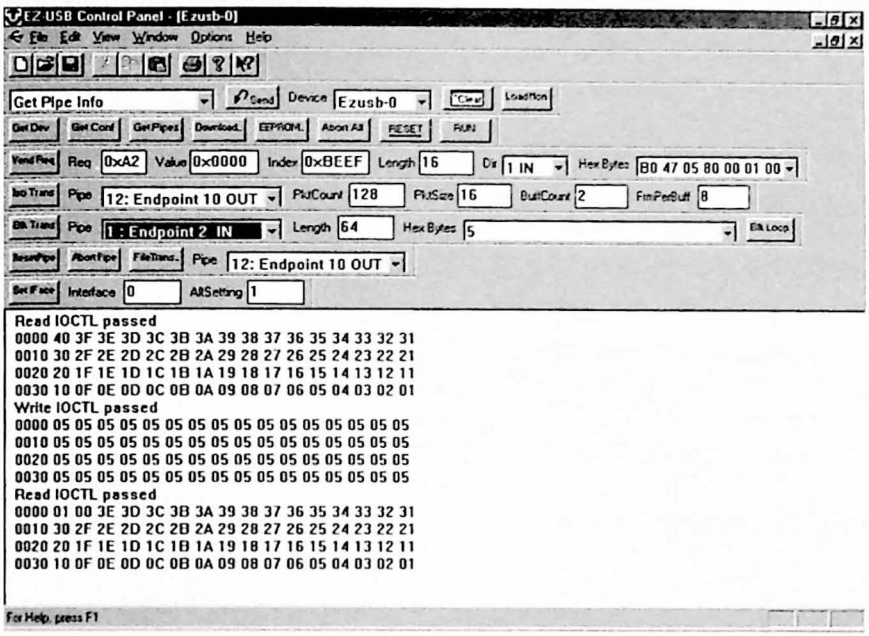


Abb. 13.3: Ein Test von Bulk-Transfers

Grundlage der hier vorgestellten Unit EZUSB2.PAS sind C-Quelltexte aus dem EZ-USB-Starterkit, speziell aus der Header-Datei ezusb.h. Die wichtigsten Treiberfunktionen sind hier in Delphi umgesetzt. Einige der Deklarationen stammen aus der Dateien usb100.h der DDK98.

Die IoControlCodes müssen in Delphi anders erzeugt werden als in C, weil es keine Möglichkeit gibt, Konstanten über eine Funktion zu definieren. Es werden daher Variablen gebildet, die über die Funktion CTL_Code initialisiert werden. Diese verbindet die vordefinierten Konstanten Devicetype, Func, Method und Access. Func ist die interne Funktionsnummer des Treibers ab $0800. In allen bisherigen Programmen wurden IoControlCodes als direkte Konstanten verwendet. Mit der neuen Unit können sie unter ihrem Namen eingesetzt werden, wie er in der Datei Ezusb.h definiert ist.

Delphi ist nicht case-sensitiv, d.h. Groß- und Kleinschreibung wird nicht unterschieden. Aus diesem Grunde musste die Funktion IOCTL_EZUSB_ANCHCHO_

DOWNLOAD in IOCTL_EZUSB_ANCHOR_DOWNLOAD_2 umbenannt werden.

Listing 13.3 zeigt Auszüge aus der neuen Unit. Der komplette Code ist im Anhang abgedruckt. Hier werden zunächst nur die notwendigen Funktionen ReadBulkData und WriteBulkData für den Bulk-Transfer gezeigt. Die Unit enthält jedoch auch alle anderen bisher eingesetzten Treiberfunktionen.

```
unit EZUSB2;

...

type _SET_INTERFACE_IN = record
 interfaceNum: Byte;
 alternateSetting: Byte;
end;

type _BULK_TRANSFER_CONTROL = record
 pipeNum: Longword;
end;

type DEVICE_TYPE = ULONG;

const Ezusb_IOCTL_INDEX  = $0800;

const
 METHOD_BUFFERED    =             0;
 METHOD_IN_DIRECT   =             1;
 METHOD_OUT_DIRECT  =             2;
 METHOD_NEITHER     =             3;

const
 FILE_ANY_ACCESS     =           0;
 FILE_READ_ACCESS    =       $0001;
 FILE_WRITE_ACCESS   =       $0002;

 FILE_DEVICE_UNKNOWN: DEVICE_TYPE = $0C0000022;

var  ...
     IOCTL_EZUSB_BULK_READ: Dword;
     IOCTL_EZUSB_BULK_WRITE: Dword;
     ...

procedure ReadBulkData (PipeNumber, NumberOfBytes: Word);
var TemplateHandle: THandle;
    DeviceHandle: THandle;    bresult: Boolean;
    nBytes: DWord;
    BulkControl: _BULK_TRANSFER_CONTROL;
    n: Integer;
```

```
begin
  BulkControl.PipeNum := PipeNumber;
  DeviceHandle := CreateFile ('\\.\ezusb-0',Generic_write,
    File_Share_write,nil,open_existing,0,TemplateHandle);
  bResult := DeviceIoControl(DeviceHandle,
     IOCTL_EZUSB_BULK_READ,BulkControl,SizeOf(BulkControl),
     @InBuffer,NumberOfBytes,nBytes,nil);
  CloseHandle (DeviceHandle);
end;

procedure WriteBulkData (PipeNumber, NumberOfBytes: Word);
var TemplateHandle: THandle;
    DeviceHandle: THandle;    bresult: Boolean;
    nBytes: DWord;
    BulkControl: _BULK_TRANSFER_CONTROL;
    n: Integer;
begin
  BulkControl.PipeNum := PipeNumber;
  DeviceHandle := CreateFile ('\\.\ezusb-0',  Generic_write,
     File_Share_write,nil,open_existing,0,TemplateHandle);
  bResult := DeviceIoControl(DeviceHandle,
     IOCTL_EZUSB_BULK_WRITE,@BulkControl,SizeOf(BulkControl),
     @OutBuffer,NumberOfBytes,
  nBytes,nil);
  CloseHandle (DeviceHandle);
end;

...

function CTL_Code (Devicetype: Dword; Func, Method,
                   Access: Word) : Dword;
Begin
...
  CTL_Code := (Devicetype shl 16) Or (Access shl 14) or
  (Func shl 2) or Method;
end;

begin
 IOCTL_EZUSB_BULK_READ := CTL_CODE(FILE_DEVICE_UNKNOWN,
     Ezusb_IOCTL_INDEX+19, METHOD_OUT_DIRECT, FILE_ANY_ACCESS);
 IOCTL_EZUSB_BULK_WRITE := CTL_CODE(FILE_DEVICE_UNKNOWN,
     Ezusb_IOCTL_INDEX+20,METHOD_IN_DIRECT,FILE_ANY_ACCESS);
...
end.
```

Listing 13.3: Die Unit EZUSB2.PAS in Auszügen

Das Programm nach Listing 13.4 zeigt einen Test des Bulk-Transfers mit den neuen Funktionen der Unit EZUSB2.PAS. Nach dem Laden der Firmware wird mit SetInterface das Interface 0 in der Einstellung 1 konfiguriert. Mit entsprechenden Schaltflächen werden Write- und Read-Transfers ausgeführt. Auch hier werden die allgemeinen Datenpuffer InBuffer und OutBuffer verwendet. Der OutBuffer wird für Sendezugriffe nicht speziell initialisiert, da es hier nur auf einen Test der Übertragung ankommt. Die 64 ersten Bytes der zufällig im Sendepuffer vorliegenden Daten werden über Pipe 2 abgesandt. Beobachtet man eine der Leitungen D+ oder D- am Oszilloskop, so erkennt man die zugehörigen Datenpakete. Anders als bei Control-Zugriffen wird jedesmal nur ein Frame benötigt. Die 64 Bytes werden also effektiv in einer Millisekunde übertragen.

Lesezugriffe mit ReadBulkData füllen den Empfangspuffer InBuffer mit 64 Bytes. Die angeforderten 64 Bytes entsprechen genau den in der Firmware abgesandten 64 Bytes. Zur Kontrolle wird hier nur das erste Byte angezeigt.

```
unit Bulk1;

interface

uses  EZUSB2,
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dia-
logs,
  StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Label3: TLabel;
    Label4: TLabel;
    Button1: TButton;
    Button2: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private-Deklarationen}
  public
    { Public-Deklarationen}
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ProgReset;
  DownLoadBin ('Bulk1.obj');
  OutBuffer[1]:= 255;
  OutBuffer[2]:= 0;
  ProgStart;
  SetInterface (0,1);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
 WriteBulkData (2,64);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
 ReadBulkData (1,64);
 Label4.Caption := FloatToStr(InBuffer[0]);
end;

end.
```

Listing 13.4: Ein einfacher Bulk-Transfer in Delphi

Über entsprechende Schaltflächen kann der Anwender OUT- und IN-Transfers testen. Die Bus-Aktivität auf den Leitungen D+ und D- kann man direkt am Oszilloskop beobachten. Es werden jeweils 64 Bytes gesendet und empfangen. Nur das erste Byte der empfangenen Daten wird angezeigt. Man sieht dann die vom Controller gesandten Zählwerte für IN-Transfers (vgl. *Abb. 13.4*).

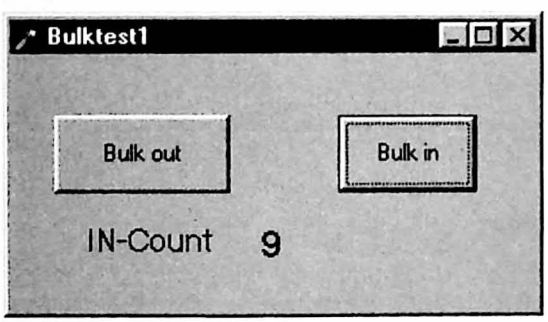


Abb. 13.4: Das Programm Bulktest1

13.4 Portzugriffe über Bulk-Transfers

Bulk-Transfers versprechen schnellere Zugriffe auf die Hardware, da jeder Zugriff nur ein Rahmen benötigt im Vergleich zu drei Rahmen für Control-Transfers. Sie lassen sich daher sinnvoll für Hardware-Zugriffe auf Ports, AD-Wandler oder andere Peripherieelemente verwenden.

Das folgende Beispiel demonstriert den Zugriff auf alle drei Ports des Prozessors. Über einen Control-Zugriff muss beim Start der Firmware festgelegt werden, welche Portanschlüsse als Ausgänge dienen sollen. Die entsprechenden Registerinhalte für die Datenrichtungsregister OEA, OEB und OEC werden von der Firmware beim Start aus den RAM-Adressen 200h bis 202h übertragen. Je nach Anwendung können bis zu 18 Ausgänge oder Eingänge verwendet werden.

Der eigentliche Bulk-Transfer ähnelt dem ersten Programmbeispiel. Hier werden allerdings nur jeweils vier Bytes übertragen. Die ersten drei Bytes repräsentieren die Zustände der Ports A bis C. Ein Byte ist unbenutzt und steht für spätere Erweiterungen bereit.

Zunächst wird der Endpoint-2-InBuffer mit den gelesenen Daten der Port-Inputregister PINSA bis PINSC gefüllt. Der erste Bulk-In-Transfer ist damit vorbereitet. Ein gleicher Programmabschnitt existiert noch einmal in der Routine serviceIN2. Sie wird immer dann aufgerufen, wenn der FIFO-Buffer ausgelesen wurde. Jedes Auslesen der Portzustände durch einen Bulk-Transfer führt also zu einer Erneuerung der Daten.

Umgekehrt werden Daten eines Bulk-Out-Zugriffs an die Portregister OUTA bis OUTC übertragen. Auch hier werden wieder nur drei der vier gesendeten Datenbytes verwendet.

```
;EZUSB, Bulktest 2
#include 8051.h

OUTA .equ   7F96h
OUTB .equ   7F97h
OUTC .equ   7F98h
PINSA .equ  7F99h
PINSB .equ  7F9Ah
PINSC .equ  7F9Bh
OEA  .equ   7F9Ch
OEB  .equ   7F9Dh
OEC  .equ   7F9Eh
dps  .equ   86h
EP2OUT   .equ 7DC0h
EP2IN    .equ 7E00h
EP2INCS  .equ 7FB8h
```

```
EP2INBC  .equ 7FB9h
EP2OUTCS .equ 7FC8h
EP2OUTBC .equ 7FC9h

start mov SP,#40h ; set stack

initPortA
      mov DPTR,#0200h
      movx a,@DPTR
      mov DPTR,#0EA
      movx @DPTR,A
initPortB
      mov DPTR,#0201h
      movx a,@DPTR
      mov DPTR,#0EB
      movx @DPTR,A
initPortC
      mov DPTR,#0202h
      movx a,@DPTR
      mov DPTR,#0EC
      movx @DPTR,A
      mov dptr,#EP2IN ; fill EP2IN buffer: PINSA to PINSC
      inc dps
      mov dptr,#PINSA
      mov r7,#3
fill  movx a,@dptr
      inc dptr
      inc dps
      movx @dptr,a
      inc dptr
      inc dps
      djnz r7,fill
      ;
      mov dptr,#EP2INBC
      mov a,#4h
      movx @dptr,a ; arm the IN2 transfer
      ;
loop  mov dptr,#EP2INCS ; Control & Status reg
      movx a,@dptr
      jnb ACC.1,serviceIN2 ; not busy, service it
      mov dptr,#EP2OUTCS
      movx a,@dptr
      jb ACC.1,loop ; busy-keep looping
      ;
serviceOUT2
      mov dptr,#OUTA
      inc dps
      mov dptr,#EP2OUT ; fill OUTA to OUTC with EP2OUT buffer
      mov r7,#3
```

```
fill2 movx a,@dptr
      inc dptr
      inc dps
      movx @dptr,a
      inc dptr
      inc dps
      djnz r7,fill2
      inc r2 ; OUT packet counter
      mov a,#4h
      mov dptr,#EP2OUTBC
      movx @dptr,a ; load bc=3 to re-arm OUT2
      sjmp loop
      ;
serviceIN2
      mov dptr,#EP2IN ; fill EP2IN buffer: PINSA to PINSC
      inc dps
      mov dptr,#PINSA
      mov r7,#3
fill3 movx a,@dptr
      inc dptr
      inc dps
      movx @dptr,a
      inc dptr
      inc dps
      djnz r7,fill3
      mov dptr,#EP2INBC
      mov a,#4h
      movx @dptr,a ; load bc=3 to re-arm IN2
      sjmp loop

.end
```

Listing 13.5: Portzugriffe über Bulk-Transfer

Die zugehörige Host-Software wurde in Delphi geschrieben. Es handelt sich hier nur um einen Test mit einer willkürlich festgelegten Festlegung von 8 Port-Ausgängen des Ports C. Das Programm läd zunächst die Firmware Bulk2.obj in den Speicher des Mikrocontrollers. Dann werden die Datenrichtungsbits festgelegt, indem die RAM-Adressen 200h bis 202h entsprechend gefüllt werden. Außerdem werden die ersten drei Bytes des Out-Puffers für den ersten BULK-OUT-Zugriff initialisiert. Dann erst wird die Firmware mit ProgStart gestartet. Der Controller führt dabei seine Initialisierung durch und legt die Datenrichtungen der Ports fest.

Zur Vorbereitung auf den eigentlichen Bulk-Transfer wird das Gerät mit SetInterface (0,1) konfiguriert. Erst jetzt stehen die benötigten Bulk-Pipes zur Verfügung. Der Bulk-Transfer selbst wird in der Timer-Routine ausgeführt und läuft immer in

beide Richtungen. Daten des OutBuffer werden an die Ports übertragen, über den InBuffer werden die aktuellen Portzustände gelesen.

```
unit Bulk2;

interface

uses  EZUSB2,
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    ScrollBar1: TScrollBar;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Timer1: TTimer;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure ScrollBar1Change(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  private
    { Private-Deklarationen}
  public
    { Public-Deklarationen}
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  ProgReset;
  DownLoadBin ('Bulk2.obj');
  WrRAM ($0200,0);   //A Inputs
  WrRAM ($0201,0);   //B Inputs
  WrRAM ($0202,255); //C Outputs
  OutBuffer[0]:= 0;
  OutBuffer[1]:= 255;
  OutBuffer[2]:= 0;
```

```
  ProgStart;
  SetInterface (0,1);
end;

procedure TForm1.ScrollBar1Change(Sender: TObject);
begin
  Label1.Caption := FloatToStr (Scrollbar1.Position);
  OutBuffer[2] := (Scrollbar1.Position);
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
 WriteBulkData (2,4);
 ReadBulkData (1,4);
 Label4.Caption := FloatToStr(InBuffer[0]);
 Label5.Caption := FloatToStr(InBuffer[1]);
 Label6.Caption := FloatToStr(InBuffer[2]);
end;

end.
```

Listing 13.6: Die Host-Software für Portzugriffe

Im Beispiel wurden nur die Portleitungen von Port C für Ausgaben initialisiert. Dabei bleiben alle Ausgaben an die Ports A und B ohne Wirkung. Nur die Daten von Port C können durch das Programm beeinflusst werden. Dazu wurde ein Schieberegler verwendet. Die eingestellten Portzustände für „Output C" werden über „Input C" auch wieder zurückgelesen. Die beiden anderen Ports dienen als Eingänge und können durch äußere Beschaltung beeinflusst werden.

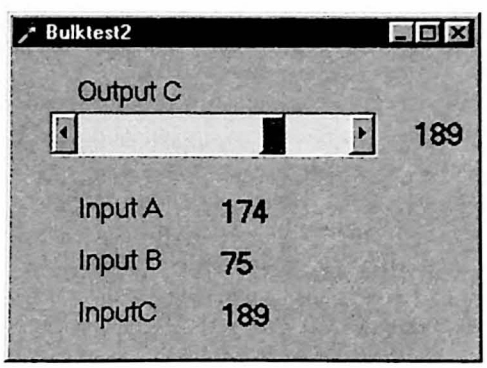


Abb. 13.5: Das Programm Bulktest2

14 Anhang

14.1 Die Delphi-Unit EZUSB2.PAS

```
unit EZUSB2;

interface
uses Windows;

type _ANCHOR_DOWNLOAD_CONTROL = record
   Offset: Word;
end;

type _GET_STRING_DESCRIPTOR_IN = record
 Index: Byte;
 LanguageId: Word;
end;

type _SET_INTERFACE_IN = record
 interfaceNum: Byte;
 alternateSetting: Byte;
end;

type _BULK_TRANSFER_CONTROL = record
 pipeNum: Longword;
end;

type _ISO_TRANSFER_CONTROL = record
 PipeNum: Longword;
 PacketSize: Longword;
 PacketCount: Longword;
 FramesPerBuffer: Longword;
 BufferCount: Longword;
end;

type _USBD_ISO_PACKET_DESCRIPTOR = record
 Offset: Longword;
 Length: Longword;
 Status: Byte ;    {....USBD_STATUS   ??}
end;

type _VENDOR_OR_CLASS_REQUEST_CONTROL = record
 direction: Byte;
 requestType: Byte;
 recepient: Byte;
 requestTypeReservedBits: Byte;
 request: Byte;
```

```
 value: Word;
 index: Word;
end;

type _USB_DEVICE_DESCRIPTOR = record
    bLength: Byte;
    bDescriptorType: Byte;
    bcdUSB: Word;
    bDeviceClass: Byte;
    bDeviceSubClass: Byte;
    bDeviceProtocol: Byte;
    bMaxPacketSize0: Byte;
    idVendor: Word;
    idProduct: Word;
    bcdDevice: Word;
    iManufacturer: Byte;
    iProduct: Byte;
    iSerialNumber: Byte;
    bNumConfigurations: Byte;
 end;

 Type _USB_ENDPOINT_DESCRIPTOR = record
     bLength: Byte;
     bDescriptorType: Byte;
     bEndpointAddress: Byte;
     bmAttributes: Byte;
     wMaxPacketSize: Word;
     bInterval: Byte;
 end;

 Type _USB_CONFIGURATION_DESCRIPTOR = record
     bLength: Byte;
     bDescriptorType: Byte;
     wTotalLength: Word;
     bNumInterfaces: Byte;
     bConfigurationValue: Byte;
     iConfiguration: Byte;
     bmAttributes: Byte;
     MaxPower: Byte;
 end;

 type _USB_INTERFACE_DESCRIPTOR = record
     bLength: Byte;
     bDescriptorType: Byte;
     bInterfaceNumber: Byte;
     bAlternateSetting: Byte;
     bNumEndpoints: Byte;
     bInterfaceClass: Byte;
     bInterfaceSubClass: Byte;
     bInterfaceProtocol: Byte;
```

```
    iInterface: Byte;
end;

type _USB_STRING_DESCRIPTOR = record
    bLength: Byte;
    bDescriptorType: Byte;
    bString: WCHAR; {  bString[1]: WCHAR;}
end;

type _USB_POWER_DESCRIPTOR = record
    bLength: Byte;
    bDescriptorType: Byte;
    bCapabilitiesFlags: Byte;
    EventNotification: Word;
    D1LatencyTime: Word;
    D2LatencyTime: Word;
    D3LatencyTime: Word;
    PowerUnit: Byte;
    D0PowerConsumption: Word;
    D1PowerConsumption: Word;
    D2PowerConsumption: Word;
end;

type _USB_COMMON_DESCRIPTOR = record
    bLength: Byte;
    bDescriptorType: Byte;
end;

type _VENDOR_REQUEST_IN = record
    bRequest : Byte;
    wValue : Word;
    wIndex : Word;
    wLength: Word;
    direction : Byte;
    bData : Byte;
end;

type DEVICE_TYPE = ULONG;

const Ezusb_IOCTL_INDEX  = $0800;

const
 METHOD_BUFFERED    =             0;
 METHOD_IN_DIRECT   =             1;
 METHOD_OUT_DIRECT  =             2;
 METHOD_NEITHER     =             3;

const
 FILE_ANY_ACCESS     =           0;
 FILE_READ_ACCESS    =       $0001;
 FILE_WRITE_ACCESS   =       $0002;

 FILE_DEVICE_UNKNOWN: DEVICE_TYPE =            $0C0000022;
```

```
var  IOCTL_Ezusb_GET_PIPE_INFO: Dword;
     IOCTL_Ezusb_GET_DEVICE_DESCRIPTOR: Dword;
     IOCTL_Ezusb_GET_CONFIGURATION_DESCRIPTOR: Dword;
     IOCTL_Ezusb_BULK_OR_INTERRUPT_WRITE: Dword;
     IOCTL_Ezusb_BULK_OR_INTERRUPT_READ: Dword;
     IOCTL_Ezusb_VENDOR_REQUEST: Dword;
     IOCTL_Ezusb_GET_CURRENT_CONFIG: Dword;
     IOCTL_Ezusb_ANCHOR_DOWNLOAD: Dword;
     IOCTL_Ezusb_ISO: Dword;
     IOCTL_Ezusb_RESET: Dword;
     IOCTL_Ezusb_RESETPIPE: Dword;
     IOCTL_Ezusb_ABORTPIPE: Dword;
     IOCTL_Ezusb_SETINTERFACE: Dword;
     IOCTL_Ezusb_GET_STRING_DESCRIPTOR: Dword;
     IOCTL_EZUSB_BULK_READ: Dword;
     IOCTL_EZUSB_BULK_WRITE: Dword;
     IOCTL_EZUSB_GET_CURRENT_FRAME_NUMBER: Dword;
     IOCTL_EZUSB_VENDOR_OR_CLASS_REQUEST: Dword;
     IOCTL_EZUSB_GET_LAST_ERROR: Dword;
     IOCTL_EZUSB_ISO_LOOPBACK: Dword;
     IOCTL_EZUSB_ISO_READ: Dword;
     IOCTL_EZUSB_ISO_WRITE: Dword;
     IOCTL_EZUSB_ANCHOR_DOWNLOAD_2: Dword;
     IOCTL_EZUSB_BULK_LATENCY_TEST: Dword;
     IOCTL_EZUSB_GET_DRIVER_VERSION: Dword;

var  InBuffer: Array [0..8191]of Byte;
     OutBuffer: Array [0..8191]of Byte;

procedure WrRAM(Adr: Word; Dat: Byte);
function RdRAM(Adr: Word): Byte;
procedure WriteRAMbytes(StartAdr, NumberOfBytes: Word);
procedure ReadRAMbytes (StartAdr, NumberOfBytes: Word);
procedure ProgStart();
procedure ProgReset();
procedure DownloadBin(FileName:  String);
procedure DownloadIntelHex (FileName: String);
procedure ReadBulkData (PipeNumber, NumberOfBytes: Word);
procedure WriteBulkData (PipeNumber, NumberOfBytes: Word);
procedure SetInterface (InterfaceNumber, Setting: Byte);

implementation

function CTL_Code (Devicetype: Dword; Func, Method,Access: Word) : Dword;
begin
  CTL_Code := (Devicetype shl 16) Or (Access shl 14) or
     (Func shl 2) or Method;
end;

procedure WrRAM(Adr: Word; Dat: Byte);       //single Byte
var TemplateHandle: THandle;
    DeviceHandle: THandle;
```

```
    bresult: Boolean;
    nBytes: DWord;
    MyRequest: _VENDOR_REQUEST_IN;
begin
   myRequest.bRequest := $A0;
   myRequest.wValue := Adr;
   myRequest.wIndex := 0;
   myRequest.wLength := 1;
   myRequest.direction := 0;
   myRequest.bData := Dat;
   DeviceHandle := CreateFile ('\\.\ezusb-0',Generic_write,
     File_Share_write,nil,open_existing,0,TemplateHandle);
   bResult := DeviceIoControl(DeviceHandle,
     IOCTL_Ezusb_VENDOR_REQUEST,@myRequest,10,nil,0,nBytes,nil);
   CloseHandle (DeviceHandle);
end;

function RdRAM(Adr: Word): Byte;
var TemplateHandle: THandle;
    DeviceHandle: THandle;    bresult: Boolean;
    nBytes: DWord;
    MyRequest: _VENDOR_REQUEST_IN;
    n: Integer;
    Buffer: Array [1..2] of Byte;
begin
   myRequest.bRequest := $A0;
   myRequest.wValue := Adr;
   myRequest.wIndex := 0;
   myRequest.wLength := 1;
   myRequest.direction := 1;     // Read
   myRequest.bData := $00;

  DeviceHandle := CreateFile ('\\.\ezusb-0', Generic_write,
     File_Share_write,nil,open_existing,0,TemplateHandle);
  bResult := DeviceIoControl(DeviceHandle,
     IOCTL_Ezusb_VENDOR_REQUEST,@myRequest,10,@Buffer,
     SizeOf(Buffer),nBytes,nil);
  CloseHandle (DeviceHandle);
  RdRAM := Buffer[1];
end;

procedure WriteRAMbytes(StartAdr, NumberOfBytes: Word);
var TemplateHandle: THandle;
    DeviceHandle: THandle;
    bresult: Boolean;
    nBytes: DWord;
    downloadControl: _ANCHOR_DOWNLOAD_CONTROL;
begin
  downloadControl.offset:=StartAdr;
  DeviceHandle := CreateFile ('\\.\ezusb-0',Generic_write,
     File_Share_write,nil,open_existing,0,TemplateHandle);
```

```
  bResult := DeviceIoControl(DeviceHandle,
     IOCTL_EZUSB_ANCHOR_DOWNLOAD_2,@downloadControl,sizeof
     (downloadControl),@OutBuffer,NumberOfBytes,nBytes,nil);
  CloseHandle (DeviceHandle);
end;

procedure ReadRAMbytes (StartAdr, NumberOfBytes: Word);
var TemplateHandle: THandle;
    DeviceHandle: THandle;    bresult: Boolean;
    nBytes: DWord;
    MyRequest: _VENDOR_REQUEST_IN;
    n: Integer;
begin
   myRequest.bRequest := $A0;
   myRequest.wValue := StartAdr;
   myRequest.wIndex := 0;
   myRequest.wLength := NumberOfBytes; //max. 1024
   myRequest.direction := 1;     // Read
   myRequest.bData := $00;
  DeviceHandle := CreateFile ('\\.\ezusb-0',  Generic_write,
     File_Share_write,nil,open_existing,0,TemplateHandle);
  bResult := DeviceIoControl(DeviceHandle,IOCTL_Ezusb_VENDOR_REQUEST,
     @myRequest,10,@InBuffer,SizeOf(InBuffer),nBytes,nil);
  CloseHandle (DeviceHandle);
end;

procedure ReadBulkData (PipeNumber, NumberOfBytes: Word);
var TemplateHandle: THandle;
    DeviceHandle: THandle;    bresult: Boolean;
    nBytes: DWord;
    BulkControl: _BULK_TRANSFER_CONTROL;
    n: Integer;
begin
  BulkControl.PipeNum := PipeNumber;
  DeviceHandle := CreateFile ('\\.\ezusb-0', Generic_write,
     File_Share_write,nil,open_existing,0,TemplateHandle);
  bResult := DeviceIoControl(DeviceHandle,IOCTL_EZUSB_BULK_READ,
     @BulkControl,SizeOf(BulkControl),@InBuffer,NumberOfBytes,
     nBytes,nil);
  CloseHandle (DeviceHandle);
end;

procedure WriteBulkData (PipeNumber, NumberOfBytes: Word);
var TemplateHandle: THandle;
    DeviceHandle: THandle;    bresult: Boolean;
    nBytes: DWord;
    BulkControl: _BULK_TRANSFER_CONTROL;
    n: Integer;
begin
  BulkControl.PipeNum := PipeNumber;
```

```
  DeviceHandle := CreateFile ('\\.\ezusb-0',Generic_write,
     File_Share_write,nil,open_existing,0,TemplateHandle);
  bResult := DeviceIoControl(DeviceHandle,
     IOCTL_EZUSB_BULK_WRITE,@BulkControl,SizeOf(BulkControl),
     @OutBuffer,NumberOfBytes,nBytes,nil);
  CloseHandle (DeviceHandle);
end;

procedure SetInterface (InterfaceNumber, Setting: Byte);
var TemplateHandle: THandle;
    DeviceHandle: THandle;    bresult: Boolean;
    nBytes: DWord;
    SetInterface : _Set_INTERFACE_IN;
    n: Integer;
begin
  SetInterface.interfaceNum := InterfaceNumber;
  SetInterface.alternateSetting := Setting;
  DeviceHandle := CreateFile ('\\.\ezusb-0',Generic_write,
     File_Share_write,nil,open_existing,0,TemplateHandle);
  bResult := DeviceIoControl(DeviceHandle,
     IOCTL_Ezusb_SETINTERFACE,@SetInterface,SizeOf(SetInterface)
     ,nil,0,nBytes,nil);
  CloseHandle (DeviceHandle);
end;

procedure ProgStart();
begin
   WrRAM($7F92,0);
end;

procedure ProgReset();
begin
   WrRAM($7F92,1);
end;

function HexToInt (Hex: String): Byte;
var h, l: Byte;
begin
  h:= ord (Hex[1])-48;
  if h>9 then h:=h-7;
  l:= ord (Hex[2])-48;
  if l>9 then l:=l-7;
  HexToInt := 16*h+l;
end;

procedure DownloadBin(FileName:  String);
vAR  f :File of Byte;
     r :Byte;
     Adresse, MaxAdresse, n, i : Word;
     Code: Byte;
```

```
begin
  AssignFile(f,FileName);
  {$I-} Reset(f); {$I+}
  r:=IOResult;
  if r = 0 then begin
    Adresse := 0;
    for n:=1 to FileSize(f)do begin;
      Read(f,Code);
      OutBuffer [Adresse] := Code;
      Adresse := Adresse + 1;
    end;
    CloseFile(f);
    WriteRAMbytes (0,Adresse)
  end;
end;

procedure DownloadIntelHex (FileName: String);
 VAR  f :TextFile;
      Line: String;
      r,wert :Byte;
      Num: Byte;
      Adr, MaxAdr, n, i, Code : Word;
begin
    AssignFile(f,FileName);
    {$I-} Reset(f); {$I+}
    r:=IOResult;
    IF r = 0 then begin
      MaxAdr :=0;
      repeat
        Readln(f,line);
       Num := HexToInt (copy(line,2,2));
        Adr := 256*HexToInt(copy(line,4,2))+HexToInt(copy(line,6,2));
        if Num>0 then begin
          for i := 1 to Num do begin
             code := HexToInt(copy(line,8+2*i,2));
             OutBuffer[Adr] := code;
             if Adr > MaxAdr then MaxAdr := Adr;
             Adr := Adr + 1;
          end;
        end;
      until Num = 0;
       CloseFile(f);
     end;
     WriteRAMbytes (0,MaxAdr+1);
  end;

begin
 IOCTL_Ezusb_GET_PIPE_INFO :=
     CTL_CODE(FILE_DEVICE_UNKNOWN,Ezusb_IOCTL_INDEX+0,
     METHOD_BUFFERED,FILE_ANY_ACCESS);;
```

```
IOCTL_Ezusb_GET_DEVICE_DESCRIPTOR :=
    CTL_CODE(FILE_DEVICE_UNKNOWN,Ezusb_IOCTL_INDEX+1,
    METHOD_BUFFERED, FILE_ANY_ACCESS);

IOCTL_Ezusb_GET_CONFIGURATION_DESCRIPTOR :=
     CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+2,
     METHOD_BUFFERED, FILE_ANY_ACCESS);

IOCTL_Ezusb_BULK_OR_INTERRUPT_WRITE :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+3,
    METHOD_BUFFERED, FILE_ANY_ACCESS);

IOCTL_Ezusb_BULK_OR_INTERRUPT_READ :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+4,
    METHOD_BUFFERED, FILE_ANY_ACCESS);

IOCTL_Ezusb_VENDOR_REQUEST :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+5,
    METHOD_BUFFERED, FILE_ANY_ACCESS);

IOCTL_Ezusb_GET_CURRENT_CONFIG :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+6,
    METHOD_BUFFERED, FILE_ANY_ACCESS);

IOCTL_Ezusb_ANCHOR_DOWNLOAD :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+7,
    METHOD_BUFFERED, FILE_ANY_ACCESS);

IOCTL_Ezusb_ISO :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+10,
    METHOD_BUFFERED, FILE_ANY_ACCESS);

IOCTL_Ezusb_RESET :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+12,
    METHOD_IN_DIRECT, FILE_ANY_ACCESS);

IOCTL_Ezusb_RESETPIPE :=
    CTL_CODE(FILE_DEVICE_UNKNOWN,Ezusb_IOCTL_INDEX+13,
    METHOD_IN_DIRECT, FILE_ANY_ACCESS);

IOCTL_Ezusb_ABORTPIPE :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+15,
    METHOD_IN_DIRECT, FILE_ANY_ACCESS);

IOCTL_Ezusb_SETINTERFACE :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+16,
    METHOD_BUFFERED, FILE_ANY_ACCESS);

IOCTL_Ezusb_GET_STRING_DESCRIPTOR :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+17,
    METHOD_BUFFERED, FILE_ANY_ACCESS);

IOCTL_EZUSB_BULK_READ :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+19,
    METHOD_OUT_DIRECT, FILE_ANY_ACCESS);
```

```
IOCTL_EZUSB_BULK_WRITE :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+20,
    METHOD_IN_DIRECT, FILE_ANY_ACCESS);

IOCTL_EZUSB_GET_CURRENT_FRAME_NUMBER :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+21,
    METHOD_BUFFERED, FILE_ANY_ACCESS);

IOCTL_EZUSB_VENDOR_OR_CLASS_REQUEST :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+22,
    METHOD_IN_DIRECT, FILE_ANY_ACCESS);

IOCTL_EZUSB_GET_LAST_ERROR :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+23,
    METHOD_BUFFERED, FILE_ANY_ACCESS);

IOCTL_EZUSB_ISO_LOOPBACK :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+24,
    METHOD_IN_DIRECT, FILE_ANY_ACCESS);

IOCTL_EZUSB_ISO_READ :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+25,
    METHOD_OUT_DIRECT, FILE_ANY_ACCESS);

IOCTL_EZUSB_ISO_WRITE :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+26,
    METHOD_IN_DIRECT, FILE_ANY_ACCESS);

IOCTL_EZUSB_ANCHOR_DOWNLOAD_2 :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+27,
    METHOD_IN_DIRECT, FILE_ANY_ACCESS);

IOCTL_EZUSB_BULK_LATENCY_TEST :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+28,
    METHOD_BUFFERED, FILE_ANY_ACCESS);

IOCTL_EZUSB_GET_DRIVER_VERSION :=
    CTL_CODE(FILE_DEVICE_UNKNOWN, Ezusb_IOCTL_INDEX+29,
    METHOD_BUFFERED, FILE_ANY_ACCESS);

end.
```

14.2 Register des AN2131

Addr	Name	Description	D7	D6	D5	D4	D3	D2	D1	D0	Default	CPU Access	Notes CPU Access Codes
	Endpoint 0-7 Data Buffers												
7B40	OUT7BUF	(64 bytes)	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R/W	RW = Read or Write,
7B80	IN7BUF	(64 bytes)	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R/W	R, r = read-only,
7BC0	OUT6BUF	(64 bytes)	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R/W	W, w = write-only
7C00	IN6BUF	(64 bytes)	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R/W	b = both (Read & Write)
7C40	OUT5BUF	(64 bytes)	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R/W	
7C80	IN5BUF	(64 bytes)	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R/W	
7CC0	OUT4BUF	(64 bytes)	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R/W	
7D00	IN4BUF	(64 bytes)	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R/W	
7D40	OUT3BUF	(64 bytes)	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R/W	
7D80	IN3BUF	(64 bytes)	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R/W	
7DC0	OUT2BUF	(64 bytes)	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R/W	
7E00	IN2BUF	(64 bytes)	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R/W	
7E40	OUT1BUF	(64 bytes)	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R/W	
7E80	IN1BUF	(64 bytes)	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R/W	
7EC0	OUT0BUF	(64 bytes)	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R/W	
7F00	IN0BUF	(64 bytes)	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R/W	
	7F40-7F5F (reserved)												
	Isochronous Data												
7F60	OUT8DATA	Endpoint 8 OUT Data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R	
7F61	OUT9DATA	Endpoint 9 OUT Data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R	
7F62	OUT10DATA	Endpoint 10 OUT Data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R	
7F63	OUT11DATA	Endpoint 11 OUT Data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R	
7F64	OUT12DATA	Endpoint 12 OUT Data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R	
7F65	OUT13DATA	Endpoint 13 OUT Data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R	
7F66	OUT14DATA	Endpoint 14 OUT Data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R	
7F67	OUT15DATA	Endpoint 15 OUT Data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R	
7F68	IN8DATA	Endpoint 8 IN Data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	W	
7F69	IN9DATA	Endpoint 9 IN Data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	W	
7F6A	IN10DATA	Endpoint 10 IN Data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	W	
7F6B	IN11DATA	Endpoint 11 IN Data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	W	
7F6C	IN12DATA	Endpoint 12 IN Data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	W	
7F6D	IN13DATA	Endpoint 13 IN Data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	W	
7F6E	IN14DATA	Endpoint 14 IN Data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	W	
7F6F	IN15DATA	Endpoint 15 IN Data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	W	

Addr	Name	Description	D7	D6	D5	D4	D3	D2	D1	D0	Default	CPU Access	Notes
	Isochronous Byte Counts												
7F70	OUT8BCH	EP8 Out Byte Count H	0	0	0	0	0	0	d9	d8	xxxxxxxx	R	
7F71	OUT8BCL	EP8 Out Byte Count L	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R	
7F72	OUT9BCH	EP9 Out Byte Count H	0	0	0	0	0	0	d9	d8	xxxxxxxx	R	
7F73	OUT9BCL	EP9 Out Byte Count L	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R	
7F74	OUT10BCH	EP10 Out Byte Count H	0	0	0	0	0	0	d9	d8	xxxxxxxx	R	
7F75	OUT10BCL	EP10 Out Byte Count L	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R	
7F76	OUT11BCH	EP11 Out Byte Count H	0	0	0	0	0	0	d9	d8	xxxxxxxx	R	
7F77	OUT11BCL	EP11 Out Byte Count L	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R	
7F78	OUT12BCH	EP12 Out Byte Count H	0	0	0	0	0	0	d9	d8	xxxxxxxx	R	
7F79	OUT12BCL	EP12 Out Byte Count L	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R	
7F7A	OUT13BCH	EP13 Out Byte Count H	0	0	0	0	0	0	d9	d8	xxxxxxxx	R	
7F7B	OUT13BCL	EP13 Out Byte Count L	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R	
7F7C	OUT14BCH	EP14 Out Byte Count H	0	0	0	0	0	0	d9	d8	xxxxxxxx	R	
7F7D	OUT14BCL	EP14 Out Byte Count L	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R	
7F7E	OUT15BCH	EP15 Out Byte Count H	0	0	0	0	0	0	d9	d8	xxxxxxxx	R	
7F7F	OUT15BCL	EP15 Out Byte Count L	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R	
		7F80-7F91 (reserved)											
	CPU Registers												
7F92	CPUCS	Control & Status	rv3	rv2	rv1	rv0	0	0	CLK24OE	8051RES	( )0011	rrrrrrbr	rv[3..0] = chip rev
7F93	PORTACFG	Port A Configuration	RxD1out	RxD0out	FRD	FWR	CS	OE	T1out	T0out	00000000	RW	0=port, 1=alt function
7F94	PORTBCFG	Port B Configuration	T2OUT	INT6	INT5	INT4	TxD1	RxD1	T2EX	T2	00000000	RW	0=port, 1=alt function
7F95	PORTCCFG	Port C Configuration	RD	WR	T1	T0	INT1	INT0	TxD0	RxD0	00000000	RW	0=port, 1=alt function
	Input-Output Port Registers												
7F96	OUTA	Output Register A	OUTA7	OUTA6	OUTA5	OUTA4	OUTA3	OUTA2	OUTA1	OUTA0	00000000	RW	
7F97	OUTB	Output Register B	OUTB7	OUTB6	OUTB5	OUTB4	OUTB3	OUTB2	OUTB1	OUTB0	00000000	RW	
7F98	OUTC	Output Register C	OUTC7	OUTC6	OUTC5	OUTC4	OUTC3	OUTC2	OUTC1	OUTC0	00000000	RW	
7F99	PINSA	Port Pins A	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	xxxxxxxx	R	
7F9A	PINSB	Port Pins B	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	xxxxxxxx	R	
7F9B	PINSC	Port Pins C	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	xxxxxxxx	R	
7F9C	OEA	Output Enable A	OEA7	OEA6	OEA5	OEA4	OEA3	OEA2	OEA1	OEA0	00000000	RW	0=off, 1=drive
7F9D	OEB	Output Enable B	OEB7	OEB6	OEB5	OEB4	OEB3	OEB2	OEB1	OEB0	00000000	RW	0=off, 1=drive
7F9E	OEC	Output Enable C	OEC7	OEC6	OEC5	OEC4	OEC3	OEC2	OEC1	OEC0	00000000	RW	0=off, 1=drive
7F9F		(reserved)											
	Isochronous Control/Status Registers												
7FA0	ISOERR	ISO OUT Endpoint Error	ISO15ERR	ISO14ERR	ISO13ERR	ISO12ERR	ISO11ERR	ISO10ERR	ISO9ERR	ISO8ERR	xxxxxxxx	R	
7FA1	ISOCTL	Isochronous Control	*	*	*	*	PPSTAT	MBZ	MBZ	ISODISAB	0000x000	rrrrrbbb	"MBZ" = Must Be Zero
7FA2	ZBCOUT	Zero Byte Count bits	EP15	EP14	EP13	EP12	EP11	EP10	EP9	EP8	xxxxxxxx	R	
7FA3		(reserved)											
7FA4		(reserved)											
	I^2C Registers												
7FA5	I2CS	Control & Status	START	STOP	LASTRD	ID1	ID0	BERR	ACK	DONE	000xx000	bbbrrrrr	
7FA6	I2DAT	Data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	RW	
7FA7		(reserved)											

Addr	Name	Description	D7	D6	D5	D4	D3	D2	D1	D0	Default	CPU Access	Notes
	Interrupts												
7FA8	IVEC	Interrupt Vector	0	IV4	IV3	IV2	IV1	IV0	0	0	00000000	R	
7FA9	IN07IRQ	EP1N Interrupt Request	IN7IR	IN6IR	IN5IR	IN4IR	IN3IR	IN2IR	IN1IR	IN0IR	00000000	RW	1=request
7FAA	OUT07IRQ	EP0UT Interrupt Request	OUT7IR	OUT6IR	OUT5IR	OUT4IR	OUT3IR	OUT2IR	OUT1IR	OUT0IR	xxxxxxxx	RW	1=request
7FAB	USBIRQ	USB Interrupt Request	*	*	*	URESIR	SUSPIR	SUTOKIR	SOFIR	SUDAVIR	00000000	RW	1=request
7FAC	IN07IEN	EP0-7IN Int Enables	IN7IEN	IN6IEN	IN5IEN	IN4IEN	IN3IEN	IN2IEN	IN1IEN	IN0IEN	00000000	RW	1=enabled
7FAD	OUT07IEN	EP0-7OUT Int Enables	OUT7IEN	OUT6IEN	OUT5IEN	OUT4IEN	OUT3IEN	OUT2IEN	OUT1IEN	OUT0IEN	00000000	RW	1=enabled
7FAE	USBIEN	USB Int Enables	*	*	*	URESIE	SUSPIE	SUTOKIE	SOFIE	SUDAVIE	00000000	RW	1=enabled
7FAF	USBBAV	Breakpoint & Autovector	*	*	*	*	BREAK	BPPULSE	BPEN	AVEN	xxxxxx00	RW	1=enabled
7FB0		(reserved)											
7FB1		(reserved)											
7FB2	BPADDRH	Breakpoint Address H	A15	A14	A13	A12	A11	A10	A9	A8	00000000	RW	
7FB3	BPADDRL	Breakpoint Address L	A7	A6	A5	A4	A3	A2	A1	A0	00000000	RW	
	Bulk Endpoints 0-7												
7FB4	EP0CS	Control & Status	*	*	*	*	OUTBSY	INBSY	HSNAK	EP0STALL	00001000	rrrrrrbb	For EP0IN and EP0OUT
7FB5	IN0BC	Byte Count	*	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	RW	* this bits are random
7FB6	IN1CS	Control & Status	*	*	*	*	*	*	in1bsy	in1stl	00000000	rrrrrrrb	at power-on. Once
7FB7	IN1BC	Byte Count	*	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	RW	operational, these bits
7FB8	IN2CS	Control & Status	*	*	*	*	*	*	in2bsy	in2stl	00000000	rrrrrrrb	read as zeros.
7FB9	IN2BC	Byte Count	*	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	RW	
7FBA	IN3CS	Control & Status	*	*	*	*	*	*	in3bsy	in3stl	00000000	rrrrrrrb	
7FBB	IN3BC	Byte Count	*	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	RW	
7FBC	IN4CS	Control & Status	*	*	*	*	*	*	in4bsy	in4stl	00000000	rrrrrrrb	
7FBD	IN4BC	Byte Count	*	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	RW	
7FBE	IN5CS	Control & Status	*	*	*	*	*	*	in5bsy	in5stl	00000000	rrrrrrrb	
7FBF	IN5BC	Byte Count	*	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	RW	
7FC0	IN6CS	Control & Status	*	*	*	*	*	*	in6bsy	in6stl	00000000	rrrrrrrb	
7FC1	IN6BC	Byte Count	*	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	RW	
7FC2	IN7CS	Control & Status	*	*	*	*	*	*	in7bsy	in7stl	00000000	rrrrrrrb	
7FC3	IN7BC	Byte Count	*	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	RW	
7FC4		(reserved)											
7FC5	OUT0BC	Byte Count	*	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	RW	
7FC6	OUT1CS	Control & Status	*	*	*	*	*	*	out1bsy	out1stl	00000010	rrrrrrrb	
7FC7	OUT1BC	Byte Count	*	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	RW	
7FC8	OUT2CS	Control & Status	*	*	*	*	*	*	out2bsy	out2stl	00000010	rrrrrrrb	
7FC9	OUT2BC	Byte Count	*	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	RW	
7FCA	OUT3CS	Control & Status	*	*	*	*	*	*	out3bsy	out3stl	00000010	rrrrrrrb	
7FCB	OUT3BC	Byte Count	*	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	RW	
7FCC	OUT4CS	Control & Status	*	*	*	*	*	*	out4bsy	out4stl	00000010	rrrrrrrb	
7FCD	OU4TBC	Byte Count	*	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	RW	
7FCE	OUT5CS	Control & Status	*	*	*	*	*	*	out5bsy	out5stl	00000010	rrrrrrrb	
7FCF	OUT5BC	Byte Count	*	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	RW	
7FD0	OUT6CS	Control & Status	*	*	*	*	*	*	out6bsy	out6stl	00000010	rrrrrrrb	
7FD1	OUT6BC	Byte Count	*	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	RW	
7FD2	OUT7CS	Control & Status	*	*	*	*	*	*	out7bsy	out7stl	00000010	rrrrrrrb	
7FD3	OUT7BC	Byte Count	*	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	RW	

Addr	Name	Description	D7	D6	D5	D4	D3	D2	D1	D0	Default	CPU Access	Notes
	Global USB Registers												
7FD4	SUDPTRH	Setup Data Ptr H	A15	A14	A13	A12	A11	A10	A9	A8	xxxxxxxx	RW	
7FD5	SUDPTRL	Setup Data Ptr L	A7	A6	A5	A4	A3	A2	A1	A0	xxxxxxxx	RW	
7FD6	USBCS	USB Control & Status	WakeSRC	*	*	*	DisCon	DiscOE	ReNum	SIGRSUME	00000100	brrrbbbb	Clear b7 by writing "1"
7FD7	TOGCTL	Toggle Control	Q	S	R	IO	0	EP2	EP1	EP0	xxxxxxxx	rbbbbbbb	
7FD8	USBFRAMEL	Frame Number L	FC7	FC6	FC5	FC4	FC3	FC2	FC1	FC0	xxxxxxxx	R	
7FD9	USBFRAMEH	Frame Number H	0	0	0	0	0	FC10	FC9	FC8	xxxxxxxx	R	
7FDA		(reserved)											
7FDB	FNADDR	Function Address	0	FA6	FA5	FA4	FA3	FA2	FA1	FA0	xxxxxxxx	R	
7FDC		(reserved)											
7FDD	USBPAIR	Endpoint Control	ISOsend0	*	PR6OUT	PR4OUT	PR2OUT	PR6IN	PR4IN	PR2IN	0x000000	RW	PRx = 1 to pair EP
7FDE	IN07VAL	Input Endpoint 0-7 valid	IN7VAL	IN6VAL	IN5VAL	IN4VAL	IN3VAL	IN2VAL	IN1VAL	1	01010111	RW	VAL =1 means valid
7FDF	OUT07VAL	Output Endpoint 0-7 valid	OUT7VAL	OUT6VAL	OUT5VAL	OUT4VAL	OUT3VAL	OUT2VAL	OUT1VAL	1	01010101	RW	VAL =1 means valid
7FE0	INISOVAL	Input EP 8-15 valid	IN15VAL	IN14VAL	IN13VAL	IN12VAL	IN11VAL	IN10VAL	IN9VAL	IN8VAL	00000111	RW	VAL =1 means valid
7FE1	OUTISOVAL	Output EP 8-15 valid	OUT15VAL	OUT14VAL	OUT13VAL	OUT12VAL	OUT11VAL	OUT10VAL	OUT9VAL	OUT8VAL	00000111	RW	VAL =1 means valid
7FE2	FASTXFR	Fast Transfer Mode	FISO	FBLK	RPOL	RMOD1	RMOD0	WPOL	WMOD1	WMOD0	xxxxxxxx	RW	
7FE3	AUTOPTRH	Auto-Pointer H	A15	A14	A13	A12	A11	A10	A9	A8	xxxxxxxx	RW	
7FE4	AUTOPTRL	Auto-Pointer L	A7	A6	A5	A4	A3	A2	A1	A0	xxxxxxxx	RW	
7FE5	AUTODATA	Auto Pointer Data	D7	D6	D5	D4	D3	D2	D1	D0	xxxxxxxx	RW	
7FE6		(reserved)											
7FE7		(reserved)											
	Setup Data												
7FE8	SETUPDAT	8 bytes of SETUP data	d7	d6	d5	d4	d3	d2	d1	d0	xxxxxxxx	R	
	Isochronous FIFO Sizes												
7FF0	OUT8ADDR	Endpt 8 OUT Start Addr	A9	A8	A7	A6	A5	A4	0	0	xxxxxxxx	RW	
7FF1	OUT9ADDR	Endpt 9 OUT Start Addr	A9	A8	A7	A6	A5	A4	0	0	xxxxxxxx	RW	
7FF2	OUT10ADDR	Endpt 10 OUT Start Addr	A9	A8	A7	A6	A5	A4	0	0	xxxxxxxx	RW	
7FF3	OUT11ADDR	Endpt 11 OUT Start Addr	A9	A8	A7	A6	A5	A4	0	0	xxxxxxxx	RW	
7FF4	OUT12ADDR	Endpt 12 OUT Start Addr	A9	A8	A7	A6	A5	A4	0	0	xxxxxxxx	RW	
7FF5	OUT13ADDR	Endpt 13 OUT Start Addr	A9	A8	A7	A6	A5	A4	0	0	xxxxxxxx	RW	
7FF6	OUT14ADDR	Endpt 14 OUT Start Addr	A9	A8	A7	A6	A5	A4	0	0	xxxxxxxx	RW	
7FF7	OUT15ADDR	Endpt 15 OUT Start Addr	A9	A8	A7	A6	A5	A4	0	0	xxxxxxxx	RW	
7FF8	IN8ADDR	Endpt 8 IN Start Addr	A9	A8	A7	A6	A5	A4	0	0	xxxxxxxx	RW	
7FF9	IN9ADDR	Endpt 9 IN Start Addr	A9	A8	A7	A6	A5	A4	0	0	xxxxxxxx	RW	
7FFA	IN10ADDR	Endpt 10 IN Start Addr	A9	A8	A7	A6	A5	A4	0	0	xxxxxxxx	RW	
7FFB	IN11ADDR	Endpt 11 IN Start Addr	A9	A8	A7	A6	A5	A4	0	0	xxxxxxxx	RW	
7FFC	IN12ADDR	Endpt 12 IN Start Addr	A9	A8	A7	A6	A5	A4	0	0	xxxxxxxx	RW	
7FFD	IN13ADDR	Endpt 13 IN Start Addr	A9	A8	A7	A6	A5	A4	0	0	xxxxxxxx	RW	
7FFE	IN14ADDR	Endpt 14 IN Start Addr	A9	A8	A7	A6	A5	A4	0	0	xxxxxxxx	RW	
7FFF	IN15ADDR	Endpt 15 IN Start Addr	A9	A8	A7	A6	A5	A4	0	0	xxxxxxxx	RW	

14.3 Literatur

[1] H.J.Kelm (Hrsg.), USB 1.1, Universal Serial Bus, Franzis-Verlag, 2. Auflage 2000

[2] D.Anderson, Universal Serial Bus System Architecture, Addison Wesley 1997

[3] J.Hyde, USB Design by Example, Wiley & Sons 1999

[4] USB Universal Serial Bus Specification Ver. 1.1

[5] H.-J.Berndt/B.Kainka, Messen, Steuern und Regeln mit Word und Excel, Franzis-Verlag, 2. Auflage 1999

[6] B.Kainka/H.-J.Berndt, PC-Schnittstellen unter Windows, Elektor-Verlag, 2. Auflage 2000

14.4 Bezugsadressen

Distributoren für die USB-Mikrocontroller CY7C63000 und AN2131 findet man im Internet unter:

www.cypress.com

Die Firma Modul-Bus stellt die im Buch beschriebenen USB-Interfaces CompuLAB-USB und Serai8/12-USB her. Die Lieferung der verwendeten Platinen und Mikrocontroller in kleinen Mengen ist ebenfalls möglich.

AK Modul-Bus Computer GmbH
Ferriéres Str. 20
48369 Saerbeck
Tel. 02574/8090, Fax 937129
e-mail: service@modul-bus.de
Internet: www.modul-bus.de

Sachverzeichnis

Symbole

24LC00 205
24LC256 206
74HC540 105

A

Abtastrate 163
Acknowledge 193
AD-Wandler 79, 148, 169
Alternate Setting 143
AN2131 138, 151
Analogeingang 91
Assembler 60, 62
Auflösung 107

B

Binärformat 163
Bit-Stuffing 17
Block-Write-Modus 207
Bootloader 216
Bulk-Transfer 21, 220
Bus-Adresse 21
Bus-Bandbreite 19
Bus-Powered 15, 108
Bus-Reset 20

C

Chip-Select 81
CloseHandle 64
Control-Request 70
Control-Transfer 20
CPUCS 141, 154
CreateFile 28, 29, 64
CTL_CODE 32, 225
CY7C63000 60, 79

D

D+ 57, 228
D- 57, 228
DA-Wandler 61
Datapointer 164
Datenpaket 15
Datenrichtungsbit 194
DDK98 114, 225
Delphi 101
Delphi-Unit 159
Deskriptor 21, 72, 143
Device-Deskriptor 23, 145
DeviceIoCommand 101
DeviceIoControl 28, 31, 64
Differenzsignal 15
Digitaleingang 91
DOS-Emulation 52
Download 154
DPS 164
DPTR 164
DS1620 62, 73
DW8051 138

E

EEPROM 148, 204
Endpoint 18, 21, 70, 220
Enumeration 21, 71, 143
Excel 128, 172, 191
EZ-USB 138, 151

F

FIFO 17, 71, 98
Fullspeed 14, 19, 58, 148

G

Grenzwert 130

H

Handle 147
HID 36
Hub 15, 18, 55

I

I/O-Request-Packet, IRP 27
I²C-Bus 148, 192
I2CS 194
I2DAT 194
IN-Transfer 224
INF-Datei 38, 62, 122, 123
Innenwiderstand 55, 58
Intel-Hex-Format 163
Interrupt 71, 197
Interrupt-Request 70
Interrupt-Transfer 21
Isochronous-Transfer 21, 40

K

Keramik-Resonator 62
Konfiguration 24, 221

L

L272 69
LM317L 57
Logikanalysator 163
Lowspeed 14, 58
LP2950CZ-3.3 57

M

Master 15, 192
MAX186 148, 168
Multiplexer 80, 85, 169
Multitasking 132

N

NRZI 16

O

OE-Register 141
Oszilloskop 40, 181, 228
OUT-Transfer 224

P

PCF8574 196
PINS-Register 151
Pipe 18, 221
PLL 138
Plotter 129
Polyswitch-Sicherung 20, 54, 56, 108
Port 138, 151, 154, 230
Portadresse 50
Portanschlüsse 61
Portausgabe 75
Portexpander 196
Power-OPV 69
Produkt-ID 24, 122, 144

Q

quasi-bidirektional 61

R

RAM 139
ReadFile 31
Referenzspannung 106
Relais 56
ReNumeration 216
Request 22
RISC 60

S

SCL 192
SDA 192
Self-Powered 15, 108

SIE 17, 71
Sink-Strom 61, 67
Sinusgenerator 42
Slave 192
Spannungsregler 57, 148
Strombegrenzung 107
Stromversorgung 54, 108
sukzessive Approximation 80, 169
Synchronisation 16

T

Taktimpuls 85, 174
TASM 222
Temperatursensor 62
Timer-Interrupt 86, 95
TLC1543 79
Ton-Burst 45
Treiber 27, 38, 62, 114, 145
Triggerung 186
TY-Schreiber 126

U

UDA1321 39
USB-Gameport 47
USB-Geräteklasse 32
USB-Kabel 13
USB-Maus 36
USB-Request 95
USB-Soundkarte 38
USB-Thermometer 62
USB-Versorgungsspannung 15

V

Vendor-ID 24, 122, 144
Vendor-Request 72, 121
Visual Basic 64

W

Win32-Driver-Model 27
WriteFile 30

X

XRAM 140
XY-Schreiber 128

Notizen

PC & Elektronik

Burkhard Kainka

MSR mit USB

Aus dem Inhalt

- USB-Grundlagen und Standardgeräte
- Stromversorgung aus dem USB
- Mikrocontroller CY7C63001 und AN2131 (EZ-USB)
- Ports, AD-Wandler und I²C-Bus

Quick-Info

- Universelle USB-Interfaces zum Messen, Steuern und Regeln
- Software-Entwicklung in Visual Basic und Delphi
- Messdaten-Erfassung und -Darstellung
- Einsatz fertiger MSR-Software zum USB

Seit der Einführung von Windows 98 ist der Universelle Serielle Bus (USB) die wichtigste Schnittstelle jedes modernen PCs. Viele Nachteile der früheren Vielfalt an Schnittstellen sind damit behoben.

Was für den reinen PC-Anwender eine Erleichterung ist, kann für den engagierten Hobbyanwender und für manchen Entwickler zu einem Problem werden. Auf den ersten Blick scheint der USB nicht so sehr für eigene Hardware-Experimente geeignet. Sehr einfache Interfaces, wie sie für die serielle Schnittstelle leicht aufgebaut werden konnten, sind beim USB nicht mehr möglich. Der neue Bus ist komplexer und aufwendiger: jedes USB-Gerät benötigt mindestens einen Mikrocontroller mit USB-Kern.

Dieses Buch macht den USB auch für semiprofessionelle Anwendungen nutzbar. Es bietet einfache Zugangsmöglichkeiten und wirksame Hilfestellungen und erleichtert Ihnen den ersten praktischen Kontakt mit USB. Schritt für Schritt werden Sie mit der Entwicklung USB-geeigneter Hard- und Software vertraut gemacht.

Auf der CD-ROM

- MSR-Anwenderprogramme • Infos zu USB-Komponenten
- Quelltexte

Hard- und Softwarevoraussetzungen:
PC ab Pentium, 16 MB Arbeitsspeicher, Windows 98, CD-ROM-Laufwerk

Noch mehr Informationen? Auch zu Ihrem Thema! http://www.franzis.de

ISBN 3-7723-5874-8

Euro 35,76 **DM 69,95**

Franzis'